Self-reconfiguring System-on-Chip using Linux on a Virtex-5 FPGA

Diplomarbeit

zur Erlangung des akademischen Grades Diplominformatiker

Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät II
Institut für Informatik

eingereicht von: Oswald Berthold

Betreuer: Dr. Frank Winkler
Erstgutachterin: Prof. Dr.-Ing. Beate Meffert
Zweitgutachter: Dr. David Krutz (DLR)

Berlin, 26. April 2012

Abstract

This thesis describes a study conducted in Reconfigurable Computing. Reconfigurable Computing is a concept almost as old as high-speed electronic computing itself. To explore the practical aspects, a Linux system embedded in Xilinx Virtex-5 Field-Programmable Gate Array (FPGA) fabric is set up and used to reconfigure its own periphery dynamically. This, at the same time, poses problems in system design and opens new possibilities in different application scenarios. Applications that have been under investigation are stream-processing in an Audio-over-IP system, approaches to Software-Defined Radio problems and evolutionary circuit design.

This work was conducted in 2010/11 in continuation of a term project at the Signal Processing and Pattern Recognition group at the Department of Computer Science of the Humboldt-Universität zu Berlin under the guidance of Frank Winkler and colleagues.

Contents

1 Introduction

1.1 Motivation

The basic topic of this thesis is Dynamic Partial Reconfiguration (DPR) on a Field-Programmable Gate Array (FPGA) and the basic question is, what kind of advantages and problems can be expected from the application of this technology. Dynamical reconfigurability is taken for granted in the software domain and the use of DPR in FPGA based systems is an attempt at combining the flexibility of software with the intrinsic parallelism of hardware. Consequently, its use enables significant reduction of space-, time- and energy-consumption on such systems, for example if reconfiguration is used to reorganise existing structures. In addition, self-modifying and self-replicating hardware can be realised within certain limits.

1.2 Hardware and software

Hardware is the messy sibling among the constituents of informatics (computer science). For any algorithm to effect the real world, it is necessary *to be implemented* in some form or another. Implementation requires an underlying piece of organised matter, commonly referred to as *hardware*. The dominating kind of hardware in use for practical work are electronic circuits. In other contexts, hardware can mean any ensemble of physical objects.

One of the great achievements of early work on the theoretical foundations of computing was the definition of the class of computable functions. The best-known formulation of this definition is Turing's concept of a paper machine, the Turing Machine (TM), although there exist many alternative descriptions which are equivalent by the Church-Turing Thesis. The basic TM can be extended to become a Universal Turing Machine (UTM), capable of reading a description of any particular TM and then simulating this machine.

This concept finds a physical counterpart, leaving finite storage aside, in the stored program computer. The power of this type of machine comes from the fact, that a description of one computing task can be read from memory and executed at one time and be replaced with another description of a different task at another time without changing the underlying mechanism. In this way, it is possible to use a fixed piece of hardware to operate on a multitude of problems without changing the hardware.

The exclusive definition of programs through memory has several consequences. Not only can one program be executed after another, but, during the execution of some process any missing or new functionality can be dynamically loaded into the execution environment. A common mechanism is that of Dynamically Shared Objects (DSOs) or Dynamically Loadable Libraries (DLLs) but there are others as well. For example, the Erlang programming language[1] features hot swapping of its basic functional units, of modules. This is possible because the definition of software processes (programs) is stored in memory and memory can be modified with very little restriction.

[1] http://www.erlang.org/

The aspect of using one fixed piece of hardware is considered a major milestone in the design of real computers. This clearly simplified a lot of things and computer architectures stabilised around the so-called *von Neumann* architecture [74] or modifications thereof, even if the term is a misnomer[2]. One characteristic of this architecture is the strictly serial organisation of processes, that is, the execution of algorithms. While simplifying things technically, it turns out that this architecture creates limits on the speed of computation of real computers. Still, this architecture or variants thereof are employed in the majority of current computer systems.

Noting, on the one hand, that this type of setup has drawbacks which are due to the serial decomposition of algorithms and, on the other hand requiring processing speeds that exceed the capabilities of serial machines, special purpose hardware structures are the only option. Their main advantage is the inherent true parallelism, which can be exploited to achieve very fast processing speeds indeed.

It is evident that the words "software" and "hardware" reflect the amount of intrinsic flexibility assigned to the concepts they designate. Up to some point the difference between hard- and software was found in the relative ease of manipulation of the latter. Where in changing a piece of hardware, human intervention was indispensable, for the early pioneers of computing, getting humans out of this loop was an important goal in order to establish fast and reliable electronic automatic computing. Now with Programmable Logic Devices (PLDs) and specifically with SRAM-based FPGAs this same flexibility can also be leveraged for hardware. This also means, that a physically implemented system can modify its own structure.

1.3 Structure of this work

After this short introductory part the section on fundamentals of reconfigurable systems will enumerate and clarify some basic concepts relating to the issues at hand. In particular it contains two kinds of historical views on reconfigurable systems, explanation of the terminology specific to Dynamic Partial Reconfiguration (DPR) on (Xilinx) FPGAs, discussion of automatic design techniques and signal processing systems in general. The section concludes with a list of example applications.

The following two sections document the practical work done within the scope of this thesis. First, the basic Microprocessor (MP) system is described with emphasis on the difference to a non-reconfigurable but otherwise equivalent system. This rests on the assumption that the burden of implementing the MP and its periphery is compensated for by the ease of handling high-level protocols and Input/Output (I/O) channels. A particular instance is the ICAP interface, which provides access to the configuration memory of the FPGA from within the MP system.

The third section describes in detail three contextually separated sets of experiments. These are audio processing, Software-Defined Radio (SDR) and Direct Bitstream Manipulation (DBM). Regarding audio processing, a system for transmitting locally captured or generated audio data over an Internet Protocol (IP) network is considered. A

[2] https://en.wikipedia.org/wiki/Von_Neumann_architecture

possible application is that of smart monitoring. For realising the "smart" bit, some amount of processing has to be done on the data stream. With reconfiguration, this can be done in a flexible manner in custom hardware modules. FPGAs play a vital role in broad-band communication systems, which are commonly considered in combination with software-defined protocol implementations. FPGAs act as an intermediate layer between analog front ends and the software defining the application. DPR can be employed again for increased flexibility within this layer. The section on DBM considers manipulating the configuration of the FPGA in the most direct way, at the bitstream level. This can provide immense speedups for generating new configurations or variations of existing templates. Some previous work in this area is followed up and converted into some practical experiments. This level will be demonstrated to still be attractive for the application of experimental design methodologies.

The work is concluded by a section summarising the main points of the thesis and outlining some perspectives on future work. All of the above is complemented by appendices dealing with tools, base system setup and Evolutionary Algorithms (EAs).

2 Fundamentals of reconfigurable systems

2.1 Scope of reconfigurable systems

Reconfiguration can be extended in scope far beyond mere computing devices. Material structures capable of reconfiguration under control of a mechanism (electronical, chemical, mechanical, ...) are of interest from many angles and pertain to many areas of scientific and engineering endeavors. In principle, "control by mechanism" implies the ability for self-reconfiguration in any kind of reconfigurable structure. This ability in turn is a step toward self-reproducing structures.

Often, the operating environment of technological artifacts is dynamic and hence flexibility is a desirable feature. More often than not it is not only dynamic but also uncertain so that the final precise operating conditions cannot be known prior to deployment. In some cases the environment can interact with artifacts beyond the pre-assigned interfaces (heat, vibration, high-energy particles, ...) and so introduce changes in the internal structure. Biological organisms cope with such environmental changes routinely and in a robust manner. This fact shall suffice as a motivation for importing biologically oriented concepts into the discussion.

> The 1990s saw the introduction of programmable devices, both digital (FPGAs) and analogue (FPAAs). These devices, by allowing the functionality and the structure of electronic devices to be easily altered, enabled researchers to endow circuits with some of the same versatility exhibited by biological entities and sparked a renaissance in the field of bio-inspired electronics with the birth of what is generally known as evolvable hardware [67, p. v].

The following paragraphs describe some examples of arguably reconfigurable systems that obviously transcend a narrow definition of reconfigurable computing but still bear a relation to current issues and will serve as an enrichment of the more specific cases discussed later on.

2.1.1 Self-reproducing automata

One of the earliest works on self-reconfiguration in the broadest sense was embedded in the effort of developing the theory of automata by John von Neumann and others. The aim was to apply the theory in the construction of reliable high-speed computing machines from unreliable components. The demand for the theory stemmed from prior experiences with the early one-off computers of the time. Apart from information theory, the main inspiration was indeed the organisation of biological organisms [73].

The universal elements of computing defined within automata theory, such as logical AND and NOT and the constants '0' and '1' define the lower limit of granularity of discrete automata. This relates to the meta-mathematical axiomatic problem, which means that wanting some given properties of a formal system, it is not clear a priori what axioms are necessary in order to achieve the desired properties in the most efficient way,

or to achieve them at all. Since the basic elements of a given computing architecture, the Processing Elements (PEs), directly correspond to axioms, it is likewise not clear a priori which ones ought to be chosen in order to optimally achieve the desired processing functions. To quote von Neumann on this matter,

> Any result one might reach in this [axiomatic] manner will depend quite essentially on how one has chosen to define the elementary parts. It is a commonplace of all axiomatic methods that it is very difficult to give rigorous rules as to how one should choose the elementary parts, so that whether the choice of the elements was reasonable is a matter of common sense judgement. There is no rigorous description of what choice is reasonable and what choice is not [73, p. 76].

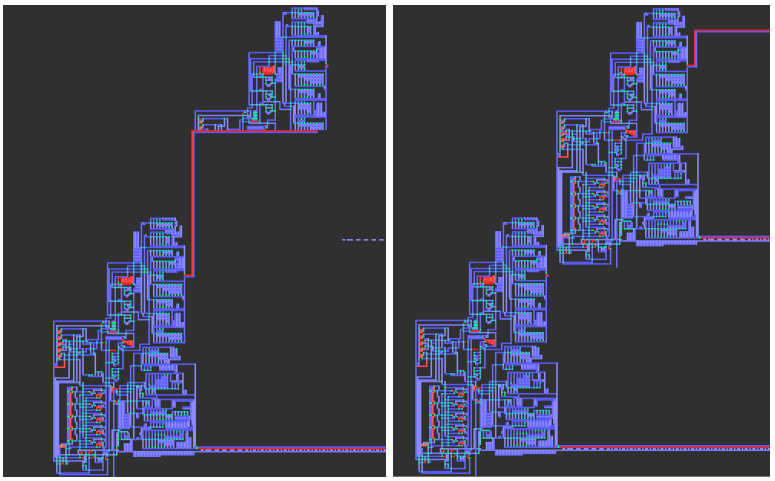
Another problem is given by the serial and parallel organisation of an entire process in regards to the logical depth of the computation. This still poses a major obstacle in algorithm development and is subject to ongoing research in automatic parallelisation and also relates to the problems of synthesis from high-level conventional programming languages [10, 15].

Finally, one particular problem which von Neumann attacked with the nascent theory of automata was the construction of a minimal automaton capable of self-reproduction. Although full self-reproducibility is not necessarily needed in the problems discussed here, it is related to problems of self-repair and it is illuminating to inspect the four most basic ingredients given by von Neumann for such a type of machine. These are

1. the neuron (or any other set of universal computing elements)
2. a muscle, the mechanics of moving parts in a sea of supplies
3. a contact maker and cutter
4. energy supply (power source)

Von Neumann arrives at a machine, realised as a 29-state cellular automaton termed the Universal Constructor, because it can not only construct a copy of itself but it can construct a copy of any structure fed to it via a tape of "genetic" instructions. An exemplary run of an implementation of the Universal Constructor is depicted in 1(a) and 1(b). The Universal Constructor is contained in the library of Golly[3], a cellular automata simulation engine. The above requirements can be mapped to the components of current FPGAs. There are basic PEs, there is a sea of parts which can be accessed, those parts can be connected via configurable routes and there is indeed an energy supply.

[3] http://golly.sourceforge.net/



(a) Partially replicated (b) Fully replicated

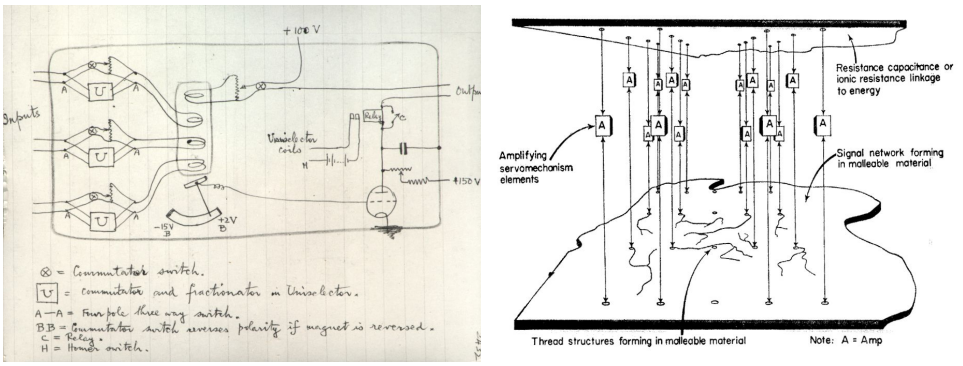
Figure 1: Screenshots of an executing von Neumann Universal Constructor in Golly. In 1(a) the automaton at the bottom is the "original" while the structure on top is a partially replicated version of the same automaton. To the right the tapes used for replication are visible. In 1(b) replication has finished and another cycle has begun.

2.1.2 Ultrastability and reconfiguration

One early reconfigurable and adaptive device that was also physically built was W. Ross Ashby's Homeostat [2]. Ashby was a British psychologist and first wave cyberneticist. He was interested in the most basic constituents of intelligent behaviour, which implied robustness and adaptivity which in turn were to be realised on the principle of homeostasis. During the end of the 1940's he constructed an analog computing machine which implements or incorporates the homeostatic principle which in that case is built upon the principle of ultrastability. Ultrastability is the ability of a system to trigger self-reconfiguration when it reaches a critical state. This is done continuously until the critical state is left after reconfiguration. This reconfiguration was accomplished in the homeostat via uniselectors. The positions of these switches could be changed electrically. The combination of four such switches with 25 positions each, spans the configuration space of one of the four components of this machine. One such component is illustrated in 2(a).

2.1.3 Pask's electro-chemical ear

Another early physical artefact, or assemblage in the original author's terms, that was conceived of in the context of Cybernetics and Artifical Intelligence was Gordon Pask's electro-chemical ear [55]. Pask's incentive, close to that of Ashby's, was to develop or find mechanisms that would correspond to principal aspects of intelligent adaptive



(a) Single homeostat unit

(b) Electro-chemical ear

Figure 2: 2(a): Structural diagram of one unit of the homeostat, taken from [1], copyright 2008 © The Estate of W. Ross Ashby. The key elements are the boxes labelled "U". These are electrically controlled switches enabling reconfiguration of the input coupling strength. 2(b): Structural diagram of Pask's sensor development experiments, image taken from [56]

systems. One of these features is the intrinsic ability of such a system to extract regularities from the environment in order to support self-perpetuation. Pask was experimenting with acidic solutions of metals and their behaviour under exposure to electric currents. If a current is passed through such a solution between two immersed electrodes, there is a tendency of growth of metallic filaments along the path the current takes which counteracts the solvent property of the acid. After many such experiments, he was able to demonstrate a device, which could dynamically develop the ability to sense certain environmental conditions, such as mechanical oscillations of certain frequencies or magnetic fields. This was achieved via a reward mechanism integrated into the experimental loop. The prevalence of one of the environmental conditions mentioned above could be read from the assemblage by probing certain electrodes within an array of electrodes for current flow. A current flowing through a given "readout" electrode indicated the detection of that stimulus by the physical structure. A schematic of this arrangement is depicted in 2(b).

> The assemblage must show a self building characteristic. If we regard the metallic thread as a decision-making device, in the sense that its presence gives rise to a current flow which selects one alternative, and its modification gives rise to a different pattern of current flow which selects another alternative, we require that if a problem is found insoluble using specified thread distribution, the assemblage will tend to build itself into a new decision making device, able to reach a solution to the problem [55, 4-13p.27].

2.1.4 Evolutionary strategy and flow problems

During the late 1950s the concept of evolutionary computation and design was put forward by different authors and further elaborated during the 1960s [23]. Leaving the pure computational (software-based) variants aside, two threads of development are of particular interest here.

Around 1967 the German computer scientist Ingo Rechenberg of Technische Universität Berlin (TUB) developed the influential Evolutionary Strategy (ES) [61] in his doctoral thesis. While interesting in many respects, one problem in particular serves to illustrate the topic at hand. Rechenberg was considering the problem of optimal flow of a liquid through a curved pipe where optimising the flow corresponds to minimising turbulence in the fluid. What he did was to set up an experiment which physically instantiated the problem, that is, circulating the fluid through the structure described. The curved piece of piping itself was made to be flexible in regard to its specific shape and this shape could be manipulated by a mechanism, that is, automatically, by a set of metal motor-controlled rods connected rigidly to the pipe. The flow through the system finally could be measured by an appropriate sensor, see 3(a).

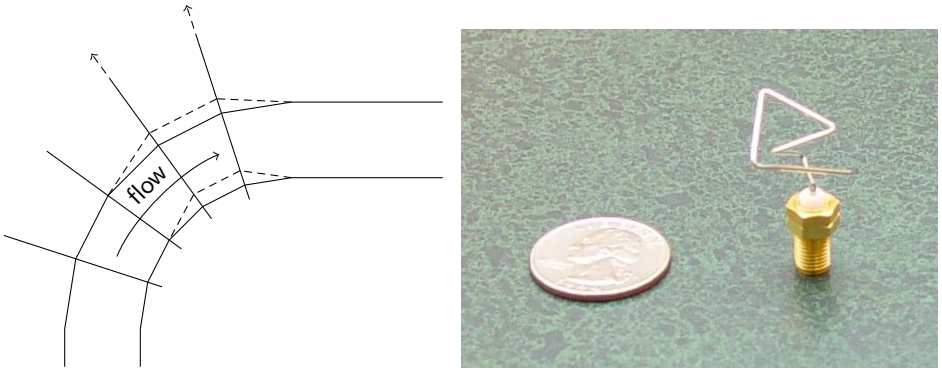
Based on this setup, Rechenberg could evaluate the ESs on the problem by programmatically configuring a given shape of the pipe, specified by a chromosome consisting of real numbers which correspond to rod positions, and determining the fitness (amount of flow) of each such configuration. Based on this fitness the evolutionary operators could be applied. This is probably the first instance of an evolutionary hardware design approach under control of a software program. As a result, surprisingly, the optimal bend shape is not symmetrical.

2.1.5 Evolutionary design of circuits, antennas and robots

There exists a large body of work on the application of Genetic Programming (GP) on electronic circuit design. GP is another variant of evolutionary computation in which the composition of executable program fragments is controlled by the genome. This has also been applied to antenna design. Almost all of this work has been based on simulation via Simulation Program with Integrated Circuit Emphasis (SPICE) or Numerical Electromagnetics Code (NEC), respectively. A comprehensive exposition of this topic is found in [22]. A prominent and recent example of this type of work is the antenna carried on-board the NASA ST-5 satellite. This antenna has been designed by a genetic algorithm through simulations. One of the key requirements that had to be met, was that the antenna had to fit into one cubic inch[4]. There are also recent examples of intrinsic evolutionary antenna design or optimisation techniques, e.g. in [36, 30].

One trend in robotics research is termed Evolutionary Robotics (ER) [49]. The approach is also based on the application of evolutionary algorithms to complete robotic systems. The simple version of this is to apply the evolutionary approach only to the software components of a robotic system. While this is relatively easy to handle it fails by far, to access the full design space of the entire system. The automatic manipulability

[4] http://www.nasa.gov/centers/ames/news/releases/2004/04_55AR.html



(a) Flow problem (b) ST-5 Antenna

Figure 3: 3(a) Flow optimisation in a 90° bend. The rods determining the bend's shape can be moved radially in- and outwards. 3(b) Evolved antenna design used on NASA's ST-5 satellites for up- and downlink.

of the robot's body and sensor integration is of key importance in that regard. Similar to all approaches seen so far, a robot body can be constructed from a small set of physical and morphological primitives that can be recombined into arbitrary shapes e.g. a walker, a snake, ring, bridges, etc. One of the current goals in *modular robotics* is the reduction in size of these primitives, likened to the multi-cellular construction of biological fabric and organs. For all sizes however, some kind of design and organisation aids are needed.

2.2 Reconfigurable computing

Returning to the reconfiguration of more conventional computing devices, reconfigurability with respect to computational processes means to be able to change the hardware, or parts of the hardware, either on a problem by problem basis or even during the lifetime of an algorithm solving one problem instance. As such, reconfigurability in software systems has robustly been accomplished with the invention of the programmable general purpose computer. This is not so for generic hardware and still somehow different for digital electronics. Nonetheless, ideas about self-reconfiguring hardware have been developed consistently throughout the history of computing [10, Ch 1] since about 1960[5] beginning with Estrin's Fixed Plus Variable architecture [20]. Initially, these concepts dealt foremost with the acceleration of specific problems, based on the realisation that the serial computational model may be computationally universal but has severe restrictions in practice. Up to now, the loadable accelerator-on-demand view can fruitfully be applied to dynamically reconfigurable systems.

[5]Even the ENIAC of 1945 could be said to be a reconfigurable computer, although this was seen more as a vice than as a virtue at the time

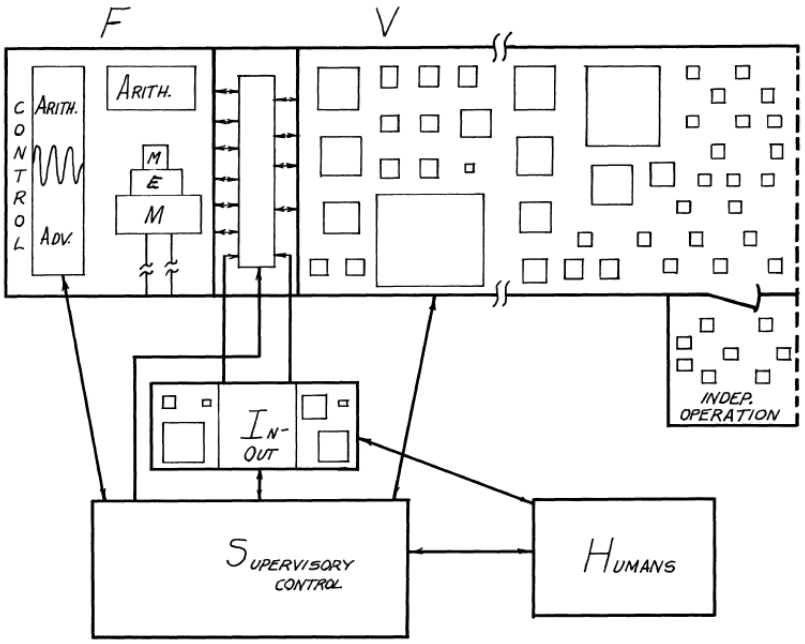


Figure 4: Structural diagram of Estrin's F+V architecture, taken from [20]. This anticipates entirely the organisation of many current reconfigurable systems.

2.2.1 Early reconfigurable computers

Fixed Plus Variable Structure Computer In about 1959 Gerald Estrin started working on a novel computing architecture in response to "Pasta's challenge" [21, p. 3] aimed at overcoming certain limitations of existing computer hardware of the time. In 1960 he published an initial proposal of how to organise such a machine [20], which states among other things that, quoting Estrin,

> The primary goal of the Fixed Plus Variable Structure Computer is:
>
> (1) To permit computations which are beyond the capabilities of present systems by providing an inventory of high-speed substructures and rules for interconnecting them such that the entire system may be temporarily distorted into a problem oriented special purpose computer.

The Fixed Plus Variable Structure Computer (F+V) consists of several high-level components.

1. The fixed part (F). This is supposed to be a classical von Neumann computer
2. The variable part (V). This is the reconfigurable area of the machine that can be populated from the inventory of high-speed substructures.

3. An I/O module

4. A supervisory control module (SC) that is to control the task decomposition and assign subtasks to F and V, respectively.

5. A routing matrix for arbitrarily connecting all of the modules

This design is highly reminiscent of current FPGA-based reconfigurable computing systems where a recent trend has been the integration of conventional microprocessors (corresponding to F) within the reconfigurable fabric (V). Obviously, data has to be passed between all components involved via a set of standardised interfaces (routing matrix). Supervisory control can be thought of as more of a logical unit rather than a physical component.

META46-GOLDLAC In 1977 Franz Rammig of Dortmund University presented the concept of an automatic hardware editor, modelled according to the operation of a text editor. The group also built a prototype of a respective support board that could implement the hardware editor's output. The main components of this system were a crossbar switching matrix and a fixed stack of library hardware modules whose in- and output connections could be configured via the switching matrix [60]. This proposal emphasises the necessity of automatic control of reconfiguration, via the editor, as well as the common property of many reconfigurable systems of having an electronically controlled connectivity matrix at their core.

Hartenstein and the XPuter Reiner Hartensteins contributions to Reconfigurable Computing (RC) are numerous. As a consequence the Xputer concept from around the second half of the 1980s is more of a meta-architecture rather than one concrete system, especially in regard to reconfiguration mechanisms. The Xputer approach does not prescribe specific implementation of reconfigurability, instead it is an approach to the organisation of hardware for a particular class of algorithms, viz. systolisable algorithms. These generally exhibit strong regularities over large chunks of data such as in image and signal processing. At the core of an Xputer arrangement lies the reconfigurable Arithmetical Logical Unit (r-ALU), which, as the name implies, can be reconfigured to perform arbitrary transforms on its inputs. A memory scan sequence, which specifies a selection sequence of data from memory is then passed through the r-ALU. Examples of scan sequences are the video scan sequence, shuffle sequence, butterfly, and trellis sequences. The generality of the Xputer concept allows for a variety of sub-architectures which are too numerous to be recounted here. Much information about the Xputer projects can be found on the Anti Machine homepage[6] or for example in [27].

Others Many other proposals for reconfigurable architectures have been put forward since the beginning of the 1980s. Some are described in Bobda's "Introduction to

[6] http://anti-machine.org/

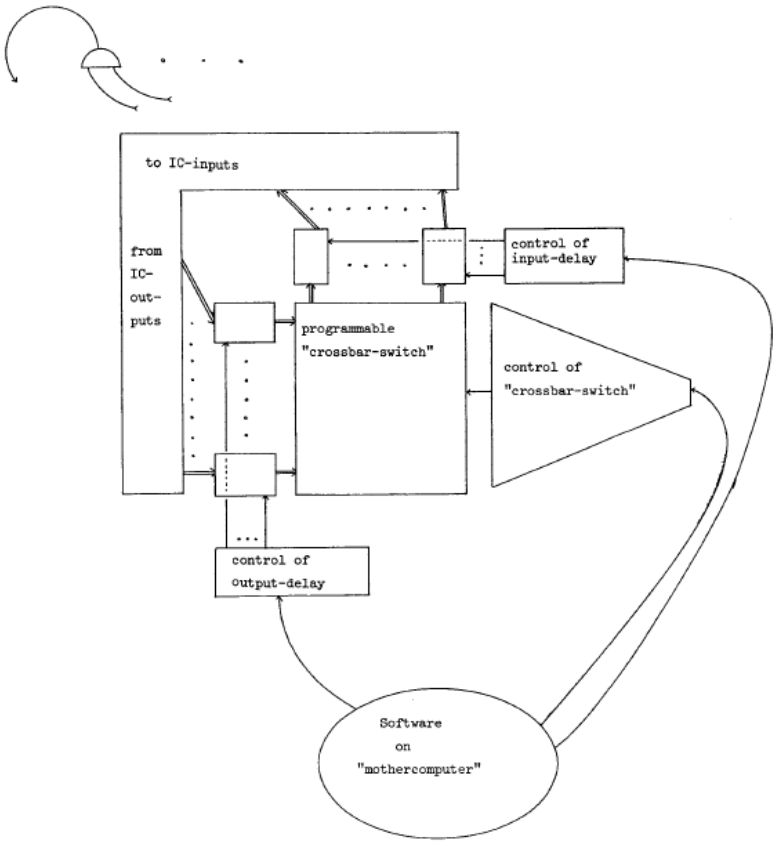


Figure 5: Structural diagram of Rammig Hardware editor system, taken from [60].

reconfigurable computing" [10]. One salient approach is that of a dynamic instruction set. This means that the compiler generates a set of specialised instructions for the current program which can then be implemented on a reconfigurable co-processor. This generally poses a problem for compilation of variable hardware/software systems from high-level languages.

Generally it can now be stated, that the problems of RC resolve into the low-level reconfiguration mechanisms on the one hand and, on the other hand, the high-level approaches taken to make use of the reconfigurability. Ideally, the configuration of hardware components (fabric primitives) and their interconnections should be fully controllable via some dynamic process. This is undoubtedly the case for current FPGAs.

2.2.2 The arrival of the FPGA

All of these earlier considerations acquire a truly new twist after the introduction of programmable logic devices in the 1980s, which led to one particularly flexible platform, viz. Field-Programmable Gate Arrays (FPGAs) or Gate Arrays in general.

Programmable Logic Devices A particular class of reconfigurable electronic devices are FPGAs, which have been mentioned several times already in passing. They represent a specific development of the more general class of Programmable Logic Devices (PLDs) which had been introduced during the 1970s. PLDs consist of two planes of fixed types of gates, often a layer of AND-gates on one plane whose outputs go to the second plane of OR-gates. Connections between device input ports to AND-gates and AND-gates to OR-gates can be configured. In this way, size considerations left aside, arbitrary logical functions can be implemented in disjunctive normal form (sum of products). A related class of devices are Complex Programmable Logic Devices (CPLDs), which consist of I/O blocks and one type of logic primitives with minimal configurability. The connection patterns among the I/O blocks and the logic cells are more freely programmable than on plain PLDs.

Field-Programmable Gate Array The first simple FPGAs did not differ much from CPLDs but quickly evolved to larger overall sizes, more versatile configurable primitives as well as additional specialised primitives for memory, fast multipliers and fast I/O. While initially some different configuration technologies were used, many devices in use today are Static Random Access Memory (SRAM) based, which enables unlimited configuration write cycles. While FPGAs are large enough to contain entire microprocessor configurations, a recent trend has gone towards including one or more *hard* microprocessor cores on-chip, as one more particularly highly specialised primitive, so to say. With the most recent and near future generations of devices this trend has only become more acute.

It is in fact only with the availability of modern FPGA that reconfigurable computing in hardware has become realisable with modest efforts on off-the-shelf devices. While traditionally FPGAs have been configured "from the outside", that is via some configu-

ration logic outside the chip itself, some devices contain on-chip reconfiguration ports. This type of resource will play a major part later on in this text. Because they allow for the use of parallel computing paths and can flexibly swap processing time for configurable space, FPGA enjoy virtual hegemony in prototyping and low-volume production runs for high-throughput signal processing devices.

2.2.3 Analog and hybrid options

Almost all commercially available programmable logic as described above is targeted towards digital circuit design. This implies using primitives of Boolean logic or higher-level macros based on those primitives for the realisation of circuits operating on discrete number representations.

There is a research direction of combining analog Very Large Scale Integration (VLSI) [43, 38] and array technology to produce analog reconfigurable fabric, commonly referred to as Field-Programmable Analog Arrays (FPAAs) as is described in [19, 4]. Analog fabric can of course be combined with digital primitives in order to produce chips of a hybrid make-up. Although such devices are, with a few exceptions, not widely available on the market, they provide a very attractive future perspective for reconfigurable hybrid computing and all considerations relating to the challenges of DPR apply to these as well. The idea is to map certain sub-processes on compound analog components. This could be done in a way similar to classical analog computing approaches such as Differential Analyzer (DiffAn)-style or diffusion networks, but of course the mapping is not limited to these concepts. Examples of currently available commercial chips are the Actel Smartfusion[7] and the Cypress PSoC[8], both of which suffer from a somehow limited amount of analog on-chip resources. Other vendors offering industry grade reconfigurable analog devices are Anadigm with two different device families (dpASP, FPAA)[9] and Lattice Semi with the ispPAC[10]. The dpASP is in fact dynamically reconfigurable using a shadowed configuration memory technique. For an extended perspective on hybrid systems see [7].

2.3 Dynamic Partial Reconfiguration Terminology

In this section the very generic picture of reconfigurable hardware painted so far will be filled in with more specific details of the implementation such systems on an FPGA while clarifying some basic concepts. Figure 6 displays the difference between global and partial reconfiguration in its most basic form.

While all PLDs are configurable per definition, some are only one-time configurable, depending on the configuration technology being used, for example fuse and antifuse based, EEPROM based or RAM based. Of those that are repeatedly configurable, only some can be reconfigured *partially*, that is, only a spatially constrained area on the

[7] http://www.actel.com/products/SmartFusion/
[8] http://www.cypress.com/?id=1353
[9] http://www.anadigm.com/dpasp.asp, http://www.anadigm.com/fpaa.asp,
[10] http://www.latticesemi.com/products/maturedevices/isppac/index.cfm

device undergoes alteration while everything else is left untouched which is referred to as Partial Reconfiguration (PR).

> Partial Reconfiguration (PR) is modifying a subset of logic in an operating FPGA design by downloading a partial configuration file [79, p. 15].

If the untouched areas also remain in operation during the reconfiguration process, this is called Dynamic Partial Reconfiguration. In effect, PR is not worth so much without its dynamical enhancement except maybe for a reduction in configuration time. Finally, as already mentioned, a tiny subset of the remaining devices can alter its own setup on-chip, that is without the need of any external configuration components. This currently leaves only a subset of the Xilinx family of FPGAs as candidates, namely chips that have the ICAP resource at their disposal. Experimentally though, DPR has also been realised on plain Spartan3 devices [3] with minimal external wiring.

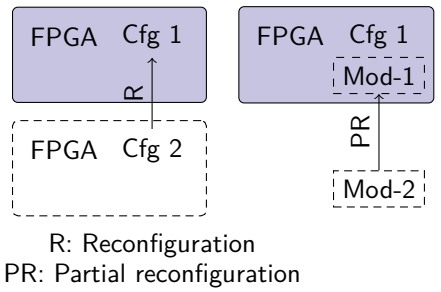


Figure 6: General depiction of a partially reconfigurable FPGA system. On the left the classical case can be seen, where the configuration for the entire chip is exchanged for a different one. During configuration, the chip is not operational. On the right a partially reconfigurable system can be seen. Only the region marked "Mod-1" is changed during reconfiguration while the logic in the rest of the chip (the static region) remains in operation.

2.3.1 Granularity

The concept of granularity has been mentioned above in passing already but it is important enough to justify more detailed discussion. Granularity refers to the size of the smallest manipulable units within a given reconfigurable medium. While this definition appears simple at first glance, several factors are interacting in the formation of a resultant system granularity. In information theoretic terms the finest possible granularity in the case of discrete systems is the binary digit (*bit*). In a real general purpose processor, on the other hand, memory might only be accessible in chunks of n bits where n is the bus width of the architecture.

The same applies to FPGAs. A bit in configuration memory is tied (hardwired) to one microscopic portion of the hardware, either for configuring the precise function of

a primitive itself or for configuring the primitives' interconnection. The configuration memory itself is made up of 32-bit chunks in the case of current Xilinx FPGAs. However, due to architectural constraints not made explicit by the vendor, configuration can only be written in portions of 41 such words for most Virtex family chips. These units are called Configuration Frames (CFs), but, they still are not at the granularity at which PR operates. The smallest reconfigurable unit, the Reconfigurable Frame (RF), in turn is made up of several CFs, 21 in the case of the Virtex-4 [5] and yet more in the case of the Virtex-5. This number also depends on the component type being configured and differs for Configurable Logic Block (CLB)-, DSP48- or Block-RAM (BRAM) blocks. If this is actually enforced by the built-in configuration logic would have to be verified by experiment. Building on top of these low-level constraints, the system can be designed to adhere to arbitrary granularities, which may not even be uniform. In general, the finer the granularity, the more efficiently the chip space can be utilized at the price of decomposition effort. More low-level details on configuration follow in the section on DBM below.

2.3.2 Partitions and modules

Partitions and modules are tied to granularity. In practical use they correlate to the granularity chosen for a specific system. A partition is an area on the chip that is reserved for reconfiguration. These are important during circuit implementation as they create zones of interdiction for the placement and routing of parts of the static system.

A module is the implemented logic definition that can be loaded dynamically into such a partition during run-time. Obviously, in the simplest possible setup, modules have to be matched to a respective partition. Nesting of partitions is only a theoretical possibility at the moment. In software terms, the partition is an abstract entity defining the interface. In addition it also defines a maximum instance size. The module is the implementation of that interface which can then be instantiated in the configurabe circuit.

2.3.3 Classification of systems

As has been outlined, the range of systems that can be termed reconfigurable is quite broad. We have already restricted ourselves to discussing FPGA-based digital electronic circuits. For such systems, Williams [75] proposes a classification along three axes.

1) Who controls reconfiguration? There are at least two components involved in configuring an FPGA. One of them is the *mechanics* of how configuration bits are written into configuration memory. The other one is the actual configuration data itself, which of course pertains to 2) just below. The mechanics can be distinguished into *exo-* and /endo-/reconfigurable variants. Mechanisms can also operate concurrently. In the exo case, the system is reconfigured from some external system like a host PC or a microcontroller system. In the endo case, reconfiguration is controlled from within the

reconfigurable system itself. The hybrid denomination could refer to situations, where an endo mechanism requests bitstreams from an external source.

2) When (and how) is the configuration being generated? Configurations can be created statically at design time, meaning that all possible instances of module and placement combinations have to be pre-generated and prepared, down to the bitstream level. A more flexible option is run-time placement where pre-implemented modules are modified during loading to adapt to a different placement on the physical fabric. This is currently not supported with the vendor-specific tools but is indeed planned [79, p. 21]. Substantial research work has been invested into the investigation of this option, see for example the works of [34, 29, 5].

The most-flexible and correspondingly most challenging *when and how* is to construct modules in an entirely dynamic manner. Possible approaches are to use generated HDL, manipulation of placed and routed netlist, bitstream templates or bitstream generation from scratch. The choice here depends loosely on 3) below. The first option has the severe drawback of requiring extensive implementation times. Of the other options, only the manipulation of routed netlists is supported by vendor methodologies via the Xilinx Design Language (XDL) netlist description.

3) What is the level of configuration granularity? Here it is necessary to deviate from William's proposal as the types of granularity he suggests are covered in 2) already. Instead, the concept of granularity discussed above can be inserted here and is tied to the overall system layout. The minimum granularity on a Xilinx chip is given by the RF consisting of about 20-50 CFs, depending on the chip and enclosed primitives.

2.3.4 Interfaces and the Hardware Operating System

In general, in a modularly structured design, all modules that need to exchange data to cooperate in a larger computation also need to share common interfaces. The module granularity has an impact on the constraints imposed on the interfaces and the complexity of the interface must be balanced vis-à-vis the module's internal complexity. For the purposes of implementation of a run-time reconfigurable system transitions from static parts to Reconfigurable Partitions (RPs) and those between RP to RP are equivalent in this case, since the connecting logic is assumed to be static, although this need not strictly be so. Down to some level, a correspondence can be drawn between the static part of a reconfigurable hardware system and the Operating System (OS) concept of providing a unified set of calling conventions to diverse I/O resources.

In endo-reconfigurable systems, by definition, some parts of the system need to remain static, or at least operational, throughout the system's lifetime. These include, minimally, the Reconfiguration Logic (RL) and a special set of I/O functionality[11]. These are, within the framework developed in this thesis, generic components such as a UART, DDR2-RAM and an Ethernet port. These parts together define a good part of the

[11] Even the reconfiguration logic could be regarded as an input/output primitive.

Hardware Operating System (HOS) in this case. In addition, there will application specific I/O present in the system and contained directly within the RPs.

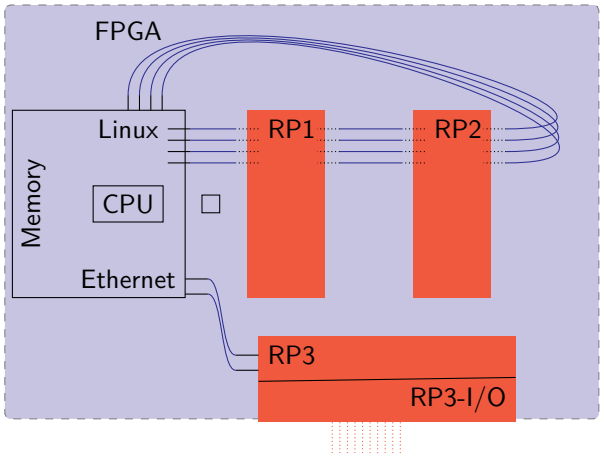


Figure 7: Static and dynamic (dynamically reconfigurable) system regions and interfaces. The blue lines indicate static interfaces. In simple cases connections between RPs are static too. The partition labelled RP3 at the bottom directly accesses FPGA I/O logic.

The HOS approach, as proposed for example in [35, 26], allows for the development of reusable PEs, irrespective of targeting PR or not. The details of using specific I/O devices such as different Analog-Digital (AD)/Digital-Analog (DA) conversion components are hidden by the surrounding HOS layer from the internal processing modules, which operate on unified I/O formats, usually streams of numbers of a given type or mixed streams of commands and data. The HOS layer needs to be changed once when the system is moved onto a different implementation platform while the hardware-independent application logic can remain unaltered. The feasibility of this approach has been demonstrated in the works cited above. Depending on the particular design methodology of a given DPR system, this approach is mandatory for a partially reconfigurable system.

2.3.5 Merit and demerit

To conclude the transition from generic reconfiguration principles to the system under investigation in this thesis, the expected advantages and problems are summarised. A distinction has to be made between basic causes and those induced by the state of technological affairs. Using PR on an FPGA system has several immediate advantages, which also provide a major source of motivation for pursuing this particular path.

Flexibility The ability to reconfigure the processing setup on demand leads to software-like flexibility in terms of system functionality. This comes without losing the intrinsic performance of custom hardware implementation as compared to serial processing on a general purpose processor and with moderate design overhead.

Resource use For some problems, a big amount of logic has to be set aside for on-chip dynamic parametrisation of a given function. Using PR this can be dispensed with, since the a temporally fixed functionality can be re-loaded with the desired parametrisation in place. This can result in the release of a substantial amount of hardware resources. Using less circuitry also means consuming less energy. This aspect can be further exploited by configuring temporarily unused partitions with placeholder modules. If overall timing constraints allow for this, computationally dependent modules can be time-multiplexed within a given partition if the necessary buffering mechanism is provided.

Insight Devising the organisation of a given complex process under consideration of reconfigurabilty enriches the design repertoire as indicated above. The challenges posed by this fact can lead to general insight into possible organisation of computing processes.

Design cycle When the HOS layer has stabilised for a given system, the design and test cycle times can be greatly reduced as compared to conventional design procedures, because implementation times of Reconfigurable Modules (RMs) are in generally much lower than when targeting the entire chip.

Novel design methods Connected with the statements from `resource use` and insight above, the PR approach in circuit design creates a mechanism for the application of non-standard design methods which is further elaborated in the next section.

There are, of course, also several problems resulting from the use of Partial Reconfiguration:

Increased initial design complexity In the initial design there is additional overhead for factoring in the technical constraints of using this particular approach.

Overall loss of spatial resource use Since the placement and routing tools are constrained by disallowed regions that are reserved for reconfiguration, the capacity for global optimisation is reduced.

Ambiguity of computational decomposition The decomposition of a given task into subtasks and primitives is not unique and is additionally complicated by the dimension of temporal placement of RMs, that is the spatio-temporal arrangement of PEs. Flexibility is a curse and blessing at the same time.

Vendor dependence Of course the details of the reconfiguration mechanism are highly vendor-dependent. This means, that if very particular features of a given mechanism are exploited on one platform, the same solution may break down on another one. This is of course also with case when accessing particular acceleration mechanisms in conventional CPUs such as in optimisation for vectorised processing.

2.4 Automated design techniques

As is the case with the HOS concept above, the use of PR lends new acuity to automated design techniques for electronic circuits. These refer to algorithms which can create other algorithms (circuit configurations) on some level of description which solve particular problems which are specified from a high-level point of view. Regarding the design complexity of large heterogenous systems, this represents an attractive enrichment of the methodologies dealing with the setup of such systems. One particular family of such "generators" which has seen the most attention in research appears to be that of evolutionary algorithms. In part this has been covered in the section on the scope of reconfigurable systems although using different methods is conceivable. The use of evolutionary methods has been covered extensively in the literature for both static optimisation as well as open-ended adaptive processes, see for example [45]. When used on hardware, the intrinsic speed of the evaluation of circuit instances can be taken advantage of. In principle, several approaches are possible.

One such approach consists of letting the EA operate on previously defined, parametrizable custom blocks, also referred to as the component approach [42]. Sometimes, virtual FPGAs are considered for this type of approach [24]. This is clearly feasible although is does not necessarily provide for the maximum parsimony [57] achievable. A second approach has been originally demonstrated by Adrian Thompson in the 1990s. In a landmark experiment he was able to use a Genetic Algorithm (GA) for configuring a subregion of a Xilinx XC6000 family chip as a frequency discriminator and as an oscillator [68, 69]. A related follow-up although not strictly FPGA-based was published in 2002 by Bird and Layzell [8]. The outstanding feature of these experiments is, that the machine-aided design process exploited chip fabric properties that are normally not included in the design space. This is termed unconstrained evolution. Combined with intrinsic evaluation the design space is greatly expanded for complex electronic setups. At the core of much of this work lies the use of switching matrices connecting at set of diverse primitives, as exemplified by the evolvable motherboard of Layzell. Yet another approach is that of Genetic Programming (GP), which is thought to be superior to GAs in many settings. There is the extensive work of Koza and others, who applied GP, among other problems, to circuit design using the SPICE circuit simulator. GP has also been used for intrinsic creation and optimisation of analog array configurations, see [4, 39]. In terms of the exploration of poorly understood computational media, evolutionary methods can help to both achieve solutions at all and to direct clarifying research. This is exemplified by the work of Spector who extended the use of GP to the exploration of programs for quantum computers [65] or by that of Harding and Miller who used EAs to explore liquid crystals for their computational capacity.

2.5 Signal processing systems

The class of applications and algorithms which are of main concern in the context of this thesis are those of signal processing. These are relevant in many application scenarios. For the implementation of such systems there are three common options, the first of which are general purpose microprocessor systems. These come in many flavours, ranging from high-end workstations using multiple processor cores over lightweight systems as are used in netbooks, mobile phones and similar devices to microcontrollers with relatively limited processing capabilities. All three differ mainly in their geometric and energy footprints. In general this approach favours flexibility over performance.

The second option are special purpose Digital Signal Processing (DSP) chips which are built for fast parallel Multiply and Accumulate (MAC) operations. A convergent tendency can be made out between DSP and general purpose serial processors. An example is the popular TI OMAP[12] package which combines an ARM microprocessor core with a DSP processor. The microprocessor thereby is enabled to delegate MAC-intensive computations to the DSP unit.

A third option for implementing such systems are FPGAs as already introduced. FPGAs consist of a fabric of basic logic elements whose functions and interconnections can be freely set by the system design. While the design overhead for these devices is higher than in standard software development, FPGAs inherently provide for highly parallel processing paths and through appropriate algorithm design immensely high processing rates can be achieved. This last aspect of FPGA makes them well suited for use in high-speed data acquisition and processing apparatus. Using Application Specific Integrated Circuits (ASICs) is not considered here.

Fabric space in an FPGA is of course a limited resource. While it is often possible to throw more hardware (larger devices, more devices) at a problem, in those cases where not all of the functionality is needed at one time, the PEs can be time-multiplexed through temporal placement. This means to decompose one computation into e.g. two steps which are executed one after another. This adds an additional dimension to the interface, viz that the output of the first module needs to be stored until the second module is loaded and ready to read the input. The overall time needed for the computation including the reconfiguration time can still be less than that needed for a fine-grained serially decomposed execution. An abstract illustration of this scheme is depicted in Figure 8.

[12] http://www.ti.com

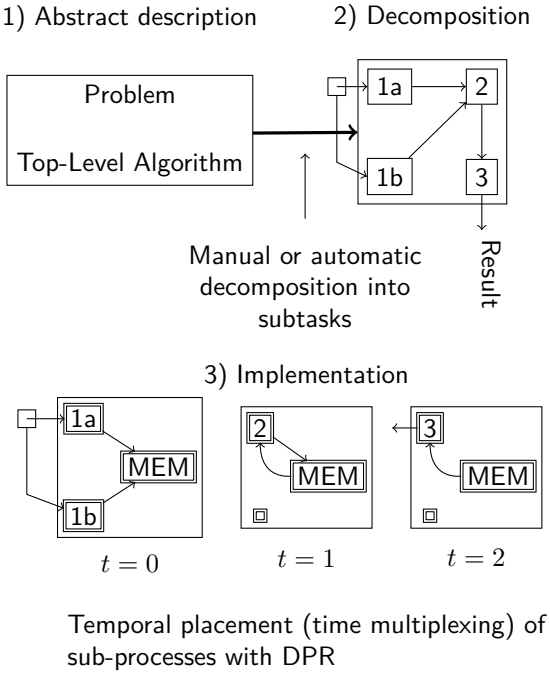


Figure 8: Temporal placement or time multiplexing of computational modules with DPR after process decomposition.

2.6 Example applications

To illustrate the usefulness of DPR in specific situations, some example applications are sketched now in more detail, extending the canonical examples from [79, 10, p. 19].

2.6.1 Audio signal processing

There are many possible applications for audio signal processing systems. These range from smart sensors and monitoring in technical and infrastructural installations or medical and health-care areas over voice communication up to musical analysis and synthesis. In a sensor node, audio processing can be used to detect events of interest and trigger either transmission of the signal itself or activation of another sensor. In multichannel setups, passive localisation of audio sources can be accomplished. Multidirectional audio communication channels can be monitored and enhanced. Finally, musical synthesis, which often requires much flexibility can be done on such a system. Audio processing has comparatively low demands on data rates and therefore serves as a well suited testbed and didactic device. Such an audio system will be described in more detail in the section on experiments.

2.6.2 Software-Defined Radio system

> "Radio in which some or all of the physical layer functions are software defined", quote from [63].

Software-Defined Radio systems are currently seeing frequent use in communication systems development and might even make the move into mass-deployment (e.g. smart phones). The basic idea in SDR is to remove as much as possible of the hardware between the antenna and the digital signal processing system. In other words, as many processing tasks as possible should be moved into the digital domain to increase the device flexibility. This applies especially to coding/decoding and modulation/demodulation of signals. A practical introduction is given for example in [31].

There are two particular challenges. One is the reorganisation of spectral allocation due to the transition from analog to digital broadcasting. The other one stems from entirely new strategies of spectrum use, for which SDR acts as an enabling agent. Two important concepts in this context are *cognitive radio* and *white spaces*. The latter are locally unused portions of the spectrum. An introduction to cognitive radio concepts along with strategies for its realisation can be found in [62].

Other challenging application areas for SDR are Ultra-Wideband (UWB) transmissions or operation on extremely weak or noisy signals such as in radiometry and radio astronomy. For all of these, it is often necessary to be able to process significant bandwidths. Since general purpose computers will not be capable of doing this in the near future, FPGAs are routinely employed in such systems.

Mobile communication shall serve as a very simple example. A smart-phone of current make (2011) uses several separate radio components for serving Universal Mobile Telecommunications System (UMTS)/Global System for Mobile Communications (GSM), Bluetooth (BT), Wireless Local Area Network (LAN) as well as Global Positioning System (GPS). A future phone might need only one reconfigurable radio module which can modify itself according to local infrastructure conditions.

2.6.3 Computation in space

On-Board spacecraft, there often is the need for low-energy, high-reliability and possibly high-bandwidth signal processing. Electronic hardware, on the other hand, travelling through space outside of the earth's protective atmosphere faces some strain, mostly from radiation but also vibrations and extreme operating temperature ranges. These combined stresses can lead to partial or complete failure of electronic devices, for example through Single Event Upsets (SEUs). DPR could be leveraged as a method for dynamic maintenance of a device's integrity. The standard measures taken are physical shielding and Triple-Mode Redundancy (TMR). These are static precaution reducing the probability of single bit errors in a calculation. Different levels of sophistication could be employed for dynamic failure detection and recovery strategies. Using naive *scrubbing* together with PR could improve overall system availability compared to global scrubbing [52]. In the past, also approaches inspired by biological cellular mechanisms

of spatial organisation and self-repair have been devised, for example see [41, 66]. Yet another approach consists of taking advantage of diversity (in terms of implementation of functionality) in populations of modules which are being modified by open-ended evolutionary algorithms [71].

2.6.4 Robotics

FPGAs are not the main implementation target for experimental robotic control systems because of challenges in development as well as the flexibility requirements. This latter point could potentially be alleviated by the application of DPR. Depending on the particular situation, FPGAs are otherwise well suited for robot development, especially for resource intensive real-time tasks such as navigation by means of visual, auditory or radio cues. They could be used on platforms of medium size without problems but on miniaturised ones they could achieve true advantages over solely microprocessor based systems. Examples of actual application of coarse-grained DPR in mobile robots are given in [14, 46].

3 Base system

Now that theoretical and conceptual aspects of dynamically reconfigurable systems have been illuminated to some extent, attention can be turned to the description of a specific experimental microprocessor System-on-Chip (SoC). This section describes the main functional characteristics of this system, while the technical details of the actual setup are delegated to an appendix. The base system makes up the static region of the reconfigurable system. It will be running Linux within the static part where its job is to mediate communications between specialised custom hardware and the outside world.

3.1 Why Linux

The Linux operating system was chosen to be run on the experimental system for several reasons. Based on the experiences with an earlier version of the network audio transmitter, it was clear that running Linux on top of the microprocessor system would lead to increased prototyping convenience. It is very well supported for the PowerPC (PPC) target as well as many other embedded targets and several meta-distributions and build-environments for embedded Linux are available [33, 51]. Using Linux, an open, robust and well documented development system can be utilised in an environment providing a rich set of debugging and additional support software, especially regarding network functionality. In consequence, it can be expected so serve as a solid platform for future developments.

3.2 Hardware platform and periphery

The target hardware that was used for almost all of the experiments described below was a Xilinx ML507[13] development board featuring a Virtex-5 FX70T FPGA. The ML507 is one the Virtex-5 family evaluation boards and as such is well equipped with diverse periphery. Relevant for the ensuing discussion are the Universal Asynchronous Receiver Transmitter (UART), Ethernet port, DDR2-RAM and audio codec. The FPGA itself comes with an embedded hard-core PPC, fast multipliers (DSP slices), and BRAM. The audio interface is provided by an Analog Devices 1981B AC97 codec chip [77], similar to the one in use on the Xilinx University Program Virtex-2 Pro (XUPV2P) board. A Marvell Alaska 88E1111 PHY controller provides physical layer access to Ethernet [77]. The RS232 port is used for debugging and early system bootstrapping.

3.3 Basic microprocessor system

Using the Xilinx Embedded Development Kit (EDK), a basic microprocessor system can be set up with the Base-System Builder (BSB) to provide the necessary foundation. This results an initial setup of the PPC, DDR2-RAM, Ethernet and UART as well as GPIO components for experimental purposes. The full details of the entire process is given in

[13] http://www.xilinx.com/products/boards-and-kits/HW-V5-ML507-UNI-G.htm

the DPR Microprocessor system appendix, while this section focusses on the structural and functional characteristics.

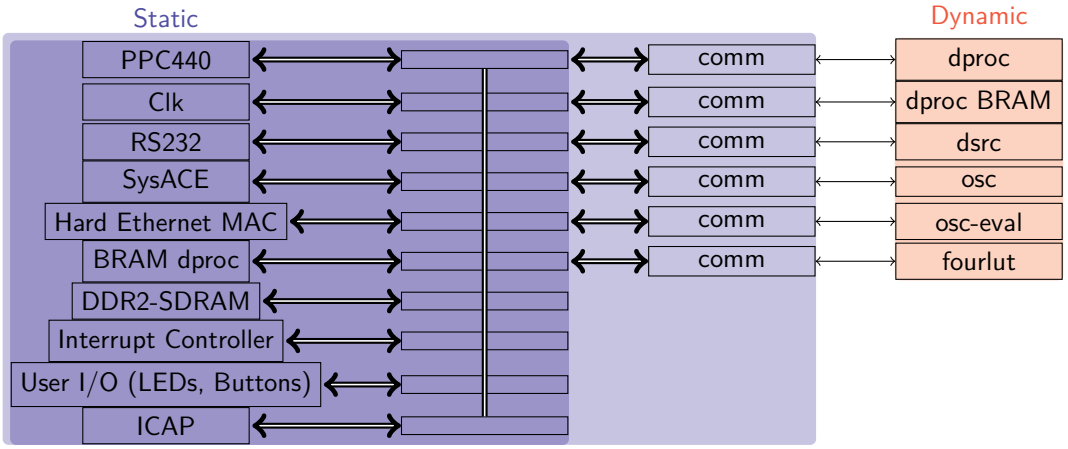
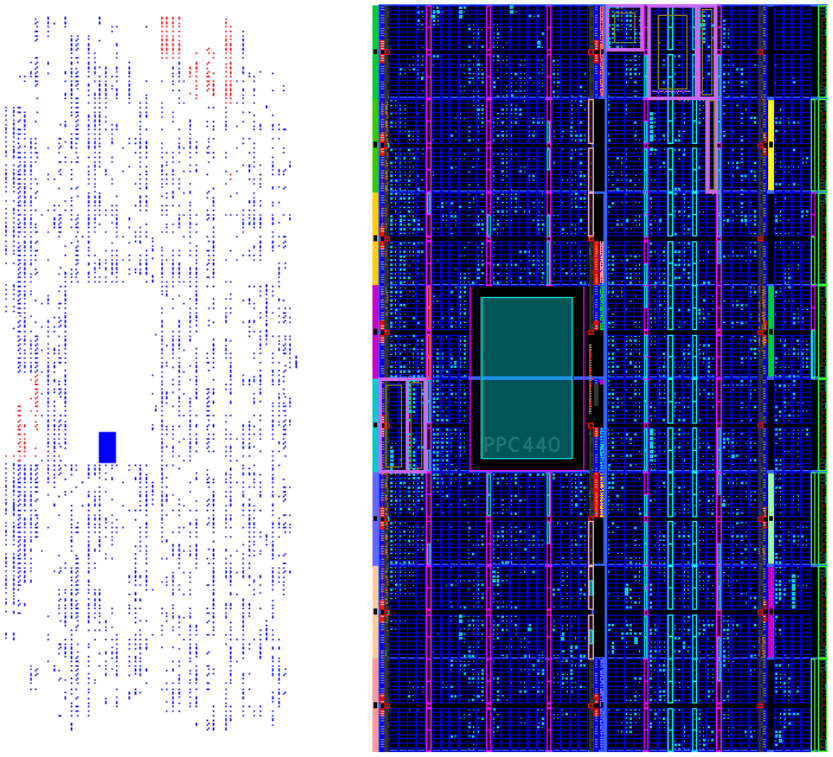


Figure 9: The overall system in the final state. The darker blue area designates the base system region, which is apart from the ICAP component identical to any standard microprocessor system.

3.4 Internal Configuration Access Port

The ICAP is the central component in the basic self-reconfigurable setup. This resource provides on-chip access to the SelectMAP configuration interface of the FPGA. It has been introduced by Xilinx with the Virtex-2. Before that, PR has already been possible via external configuration methods[14] on the first-generation Virtex chips. On the Virtex-5 FX70T there are two such components available which can in principle be operated independently. For the purposes of this work however, only one of them has been used. The ICAP can be accessed by any logic, so a microprocessor is not strictly necessary for achieving auto-reconfigurability. For reasons given above, Linux performs the interface functions and Estrin-style supervisory control so it has to be able to access the ICAP. This is provided by a combination of two components. One is the *xps_hwicap* IP-core which connects the ICAP hardware primitive to the Processor Local Bus (PLB) bus used within the microprocessor design. The other component is a Linux kernel driver module which creates a device file that can be read from and written to for configuration read back and partial reconfiguration, respectively. The simple-most example of such an access is issuing

[14] http://forums.xilinx.com/t5/Spartan-Family-FPGAs/Spartan-Family-and-ICAP-Partial-Reconfiguration/td-p/49683



(a) System view in FPGA Editor (b) System view in floorplanner

Figure 10: 10(a) Base system implemented and viewed in FPGA Editor. 10(b) Another view in the Floorplanner with reconfigurable partitions marked by the magenta frames.

```
$ cat partial-bitfile.bit >/dev/icap0
```

in the Linux system, which writes raw configuration data contained in "partial-bitfile.bit" into the configuration memory. The bitstream file is sufficient since the configuration of the reconfigurable frames is contained entirely within the bitstream.

The time needed for configuration is a linear function of the bitstream length. Quoting from the Xilinx User Guide 702 [79]:

> The speed of configuration is directly related to the size of the partial bit file and the bandwidth of the configuration port. The different configuration ports in Virtex architectures have the maximum bandwidths shown in Table 1.

Table 1: Table of bandwidths for different configuration methods

Configuration Mode	Max Clock Rate	Data Width	Maximum Bandwidth
ICAP	100 MHz	32 bit	3.2 Gbps
SelectMAP	100 MHz	32 bit	3.2 Gbps
Serial Mode	100 MHz	1 bit	100 Mbps
JTAG	66 MHz	1 bit	66 Mbps

For a minimal Reconfigurable Frame (RF) of 1×20 CLBs the raw bitstream length (without the header) is 6632 bytes. The exact number varies according to the primitives contained within this region (see the next section for details). In this example the time needed for reconfiguration of the region amounts to

$$\frac{6632 \cdot 8bit}{3.2e9bps} = 0.00001658s = 16.58us$$

The ICAP has another interesting feature, viz that it can be used to monitor any configuration process. The status of any ongoing configuration is emitted on the ICAP's "O" port automatically and can be tapped for debugging. While only the Virtex family chips (starting with the Virtex-II) and the Spartan-3A and Spartan-6 provide the ICAP resource, independent work has demonstrated the feasibility of an ICAP-equivalent setup for Spartan-3 platforms with minimal additional external wiring using either serial or parallel [3] configuration modes. Parallel modes are preferable as the are intrinsically faster. The approach described in [3] can be extended to platforms other than the Spartan-3 as well. The ICAP primitive belongs to a family of configuration logic related primitives. An interesting sibling in this family is the CAPTURE primitive (CAPTURE_VIRTEX5 in the case of Virtex-5, [76, pg 102]). This block allows for a restricted custom chip-scope implementation. When triggered by assertion of the CAP signal, it will store the current values of all CLB, BRAM and Input/Output Block (IOB) registers in their respective

places in the configuration memory, which then be read back via normal readback procedure, e.g. via the ICAP. In this way, interesting introspective and debugging functions could be implemented.

3.5 Application logic

All application logic will be described in the next section. During design the base system needs to be made aware of any RPs present within the system. This is done by instantiating IP-cores which act as wrappers to the actual RMs. These wrappers contain references to components without implementation (black-boxes), which is of no concern during synthesis. The netlists for specific RMs will be integrated into the overall design during implementation.

The interface via which the static system communicates with the instantiated RM is fixed at design time of the static part. In the experiments described below, only very simple interfaces have been used. In the course of the work described below, the initial base-system has been incrementally extended to contain specific regions used for the three main experimental applications. The final state of implementation can be seen in 10(a) and 10(b) from the chip-geometry perspective and in Figure 9 from a structural one.

There exists a respectable amount of previous work on basic platforms for self-reconfiguration. Some aspects are also shared with embedded Microprocessor systems in a completely static execution environment. As such, inspiration has been drawn especially from [11] and the Xilinx Open Source Wiki[15]. Some recent exemplary contributions with similar overall goals as those pursued in here are [58, 54, 26] but similar approaches have been proposed much earlier by Blodget [9] or Williams [75]. None however are going into details of low-level on-the-fly generation of configurations.

[15] http://xilinx.wikidot.com/

4 Experiments

4.1 Audio-Processing

The basic idea is to realise a device, possibly remotely installed, that will continuously monitor an incoming audio signal, for example from a microphone or an arbitrary analog line level signal. A process running within the device is monitoring the stream for events. These events are defined by classifiers that are implemented on the device and which operate on the input stream. One of the simplest possible such events is amplitude threshold crossing. Once such an event has been detected, any kind of action can be triggered. In this experiment, the action is to activate the transmission of the incoming signal via an IP-network. Other possible actions are activation of other sensors such as a camera or general remote signalling. The transmission system is not limited to such applications though and could also be used for example as a high-performance audio processing and synthesis engine for musical applications. The basic concept is diagrammed in Figure 11.

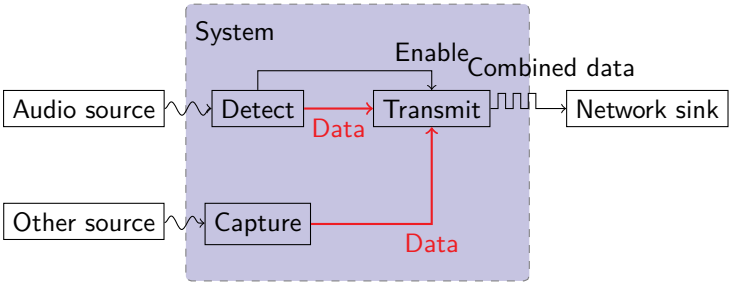


Figure 11: Basic scheme of an event based stream-transmitter.

4.1.1 Preliminary considerations

As part of a larger ongoing project at the department concerning different aspects of reliable WAN audio coding and transmission, a prototype of such a system has been developed on FPGA technology as a semester project. As this was done on an XUPV2P board, both PPC cores available on this board were utilised for interfacing to the outside world. One of the processors handles the AC97 audio codec while the second one provides UDP/IP functionality. The data-path between the two processors was implemented in custom hardware via a BRAM-based double-buffer and featured the threshold activator circuit which controlled network transmission as well as providing the data to be transmitted.

Extending this setup, the system was reimplemented under different conditions. Foremost this was done to introduce and explore the use of PR. In order to take advantage of smaller reconfiguration granularity, the ML507 was chosen for implementation. The Virtex-5 FX70T features a single PPC core. While not strictly necessary, for reasons of

compactness the AC97 codec was interfaced directly with custom hardware. In this way, the input to the processing chain was detached from the microprocessor. Additionally, the processor is run under Linux, largely facilitated by existing in-house expertise [11] of Linux on Xilinx FPGA and in the hope of leveraging the Unix tool modularity for prototyping purposes.

4.1.2 Processing chain

Audio processing is realised as a chain of modules that process stream data one after another. The simplest setup contains a source node, a stream processor, and a sink node, a corresponding diagram is depicted in Figure 12. In a linear chain, source and sink are unique whereas the intermediate processors can basically vary in number. A more versatile signal-flow network would be a logical extension for later systems, a sketch of such a setup is depicted in Figure 13.

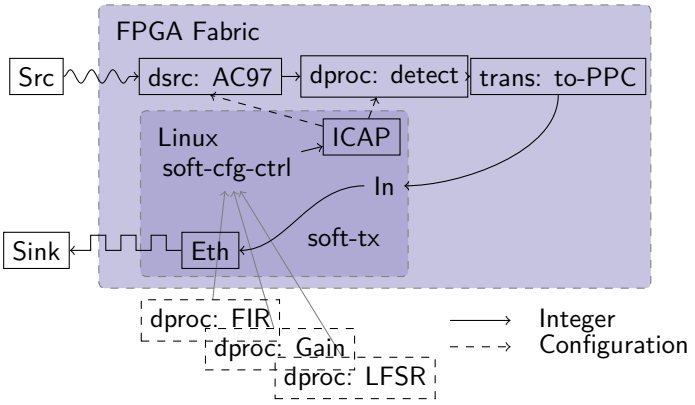


Figure 12: Detailed scheme of the event based stream-transmitter. This structure corresponds closely to the system acutally implemented.

Source nodes These are called *dsrc* in the diagrams. The primary source node is the AC'97 audio codec for capturing analog audio input signals. The source node was realised as a RP in order to enable dynamic replacement. For that reason, the audio codec readout has to be driven from custom hardware while conforming the stream interface at the output. In this way, the audio capture source can be replaced by noise generators or oscillators or, more generically, by any module conforming to the interface of the subsequent module, later on. The AC97 dsrc interface can be inspected in Figure 14.

In-Chain interface The internal interface of the processing chain consists of 32 wires for representing two channels of 16-bit Pulse-Code Modulation (PCM) data in signed integer format. One additional bit is used to represent the stream clock, which in the case of the AC'97 source is connected directly to the codec's audio clock. Momentary

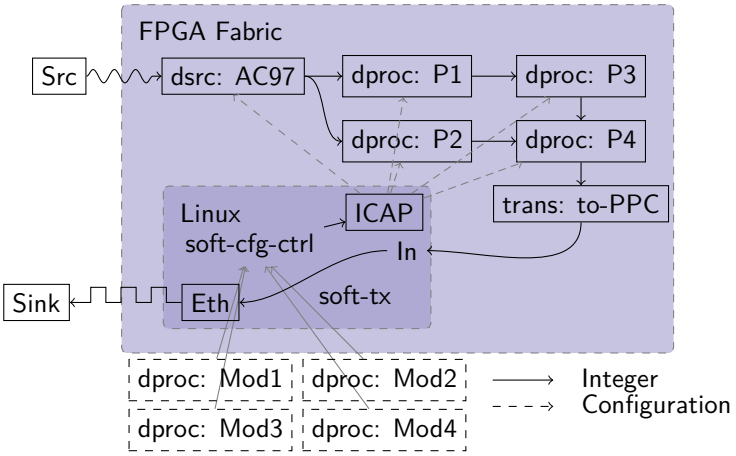


Figure 13: Detailed scheme of the event based stream-transmitter in a slightly extended version where the signal flow is not strictly linear anymore. Other extensions such as two-way communication are not drawn.

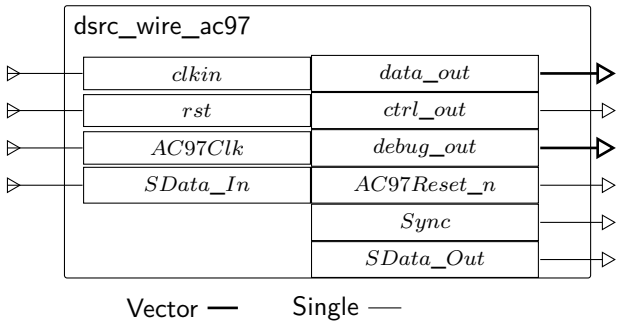


Figure 14: Simple interface for a reconfigurable data source module.

glitches are allowed in the stream so no additional provisions for inter-module handshake had to be made. The additional $ctrl\_in$ and dbg_out ports are used for control and debugging purposes, respectively, see Figure 15. Any processing module can also be thought of as a generator, depending on no or *little* input.

Sink nodes The sink node is responsible for aggregating the stream data from the sample-based hardware processing chain into blocks of data suitable for handling by the microprocessor. In principle, the chain can be designed to contain a return line, so that signals can be passed through the chain in both directions. This has been left out in the initial design for simplicity. The input of the sink node complies with the standard wire-based stream interface. The output is written into a bi-partitioned BRAM segment to enable efficient transfer to the microprocessor.

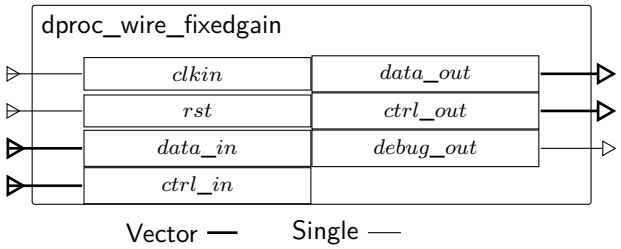


Figure 15: In-chain module interface.

For the final transfer of data from the chain into the microprocessor, again a BRAM-based mechanism was employed. The address counter is increased by the audio clock arriving via the single bit control line. The control output is toggled between two states depending on the value of the address counter. This indicates to the microprocessor, which BRAM segment is ready for transfer. The structure is depicted in Figure 16.

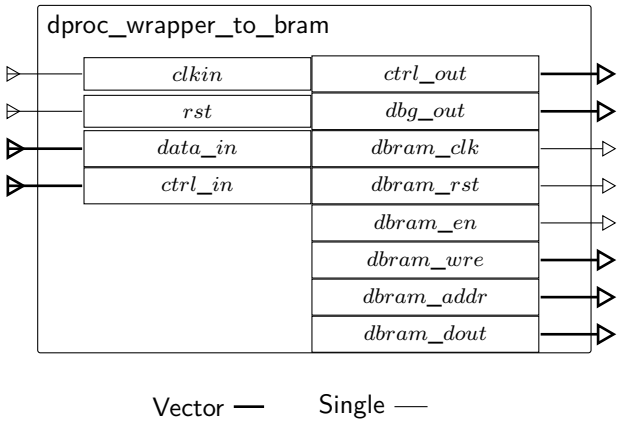


Figure 16: Sink module interface.

4.1.3 Modules

These modules are the core of the audio processing application. The current experimental setup contains only a source, one processing module and a sink. The modules implement the interface described above symmetrically on their inputs and outputs. Thus they can in principle be chained together into longer sequences. Several modules have been developed for evaluating the setup. These are described in the following paragraphs.

***identity*-module** This is used as the default configuration. The only function it performs is to copy its input to its output without modification. It is used to test the entire transmission and to verify that the input signal acquisition works.

***gain*-module** The gain module applies a fixed gain on the signal embedded in the data stream. This is a minimal example of a parameterisable module for which the logic for modifying the parameter can be saved by reconfiguration. Modules can be pregenerated with fixed gains from Hardware Description Language (HDL). Another option is to use a bitstream template in which the parameter can be replaced on-the-fly prior to reconfiguration. In the latter case, the structure has to be designed to represent the coefficients in a form suitable for such a replacement. One possibility is to use a shift-add structure for multiplication with a constant. Alternatively the multiplication table can simply be encoded on a set of Look-Up Tables (LUTs) and could hence be subject to LUT rewriting directly in the bitstream.

***Finite Impulse Response (FIR)*-filter-module** The FIR-filter module implements a fourth order FIR-filter for two channels according to the digital filter equation

$$y(n) = \sum_{i=1}^{N} b_i x(n-i)$$

The same considerations as for the gain module apply here. The filter coefficients

$$\mathbf{b} = (b_0, ..., b_4)$$

represent the parameters of this module where the same set of coefficients is used for both channels. So either the coefficients are passed into the module together with the signal or the coefficients are fixed for one implementation. In the second case, implementations for different values of the coefficients can be pregenerated or they could be replaced in the bitstream prior to reconfiguration.

***threshold*-module** The threshold module is a nonlinear filter implementing the equation

$$y(n) = \begin{cases} x(n) & \text{if } \bar{x} \geq \theta, \\ 0 & \text{if } \bar{x} < \theta. \end{cases} \text{ with } \bar{x} := \frac{1}{N} \sum_{i=0}^{N} |x(n-i)| \text{ and threshold } \theta$$

In addition, this module surpresses emission of the control signal. In this way, audio transmission is immediately suspended by means of the BRAM to network copy mechanism in software.

***Linear Feedback Shift Register (LFSR)*-module** A processing module does not necessarily have to use its signal input but can also act as a signal generator by itself. The limited sizes of the RPs ask for efficient implementation of such generative structures and the LFSR module is only meant as an example of generator instantiation. The implemented module uses a register length of 16 with a maximal cycle length of 65535 although this cycle length still results in audible tones.

4.1.4 Transmission setup

Transmission is implemented as a Linux software program. The program monitors the General Purpose Input Output (GPIO) status signal and directly transmits the respective portions of the BRAM on the network interface via UDP/IP. If the control signal is kept low for both buffers, transmission pauses. The resulting audio stream on the network channels consists of a raw stereo PCM stream over UDP which can be received and "decoded" with minimal effort. The prototype receiver is a standard Linux computer running *netcat*, which provides the listening socket. netcat's output is redirected into *aplay* via a process pipe which writes the received signal onto the local audio interface. This happens in the transmitter and receiver software modules, called *soft-tx* and *Sink*, respectively in Figure 12 and Figure 13. On the embedded system the following command initiates transmission

```
[root@molly src]# ./bram2net
```

where *bram2net* is the software transmitter. On the receiver side the stream can be played back with some standard commands, such as

```
nc -l -u -p 5000 | aplay -t raw -c 2 -f S16_LE -r 48000 -
```

Figure 17 contains a waterfall segment of a piece of music transmitted in this manner while the processing module in hardware undergoes several cycles of reconfiguration. Figure 18 zooms in on the transient during reconfiguration. The dropout amounts to approximately 20 ms which could easily be masked via different techniques.

4.1.5 Possible extensions

The audio processing system is a simple prototype whose main purpose is to demonstrate the basic mechanism of using reconfiguration in this type of setup. There are numerous options for further elaboration, some of which will be sketched below.

Processors as generators As already mentioned, processing modules can be viewed as generating systems that can be driven by external inputs. The actual coupling can vary significantly ranging from *identity*, that is, copying input to output, to *generator*, that is, an autonomous generator of periodic signals with variable properties.

If different sources are used that require different sets of capture hardware external to the FPGA, such as the audio codec, provision has to be made for the reconfigurable

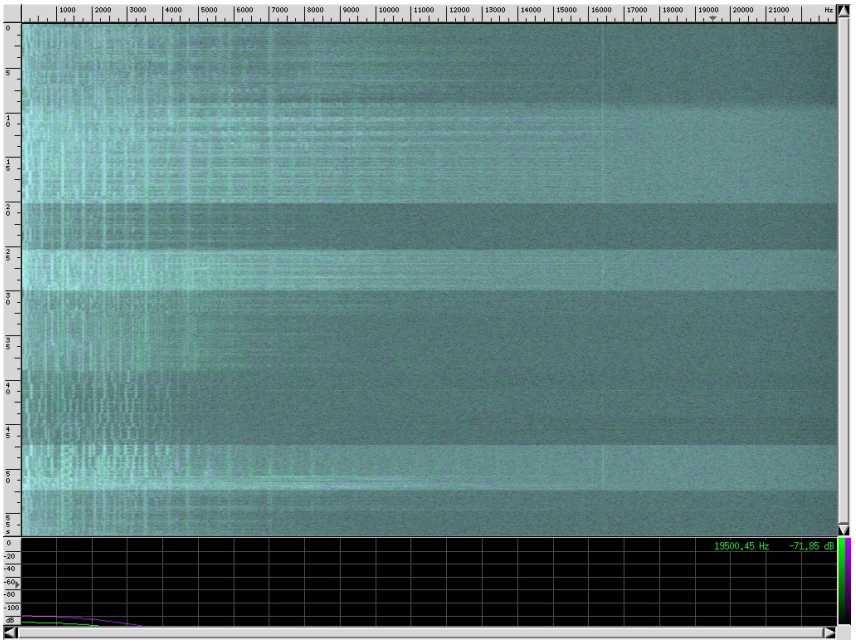


Figure 17: Waterfall display of transmitted signal containing several reconfigurations.

modules to be able to access this hardware. This functionality is supposed to be provided by the HOS layer.

Single channel operation The entire design could be cleaned up in terms of channel handling. Initially, two-channel processing was an implicit assumption but the structure can of course be modified to provide a configurable number of single channels. In addition, other parameters of a channel such as width and sampling rate should be considered to be variable.

Branching The linear chaining of processing blocks in between a single source and single sink node is a very restricted model. Indeed, it would be desirable to be able to specify signal flows with arbitrary branching. This could be realised by prepending samples with channel numbers and sharing them on a bus.

Encoding It is conceptually easy to think of modules within the processing chain or network that change the encoding of data in a way that is advantageous for later steps, such as compression or encryption. As a simple example, consider audio compression with a lossy codec prior to transmission. Clearly, this can also be dealt with in hardware while the choice of the codec is left flexible through reconfigurability.

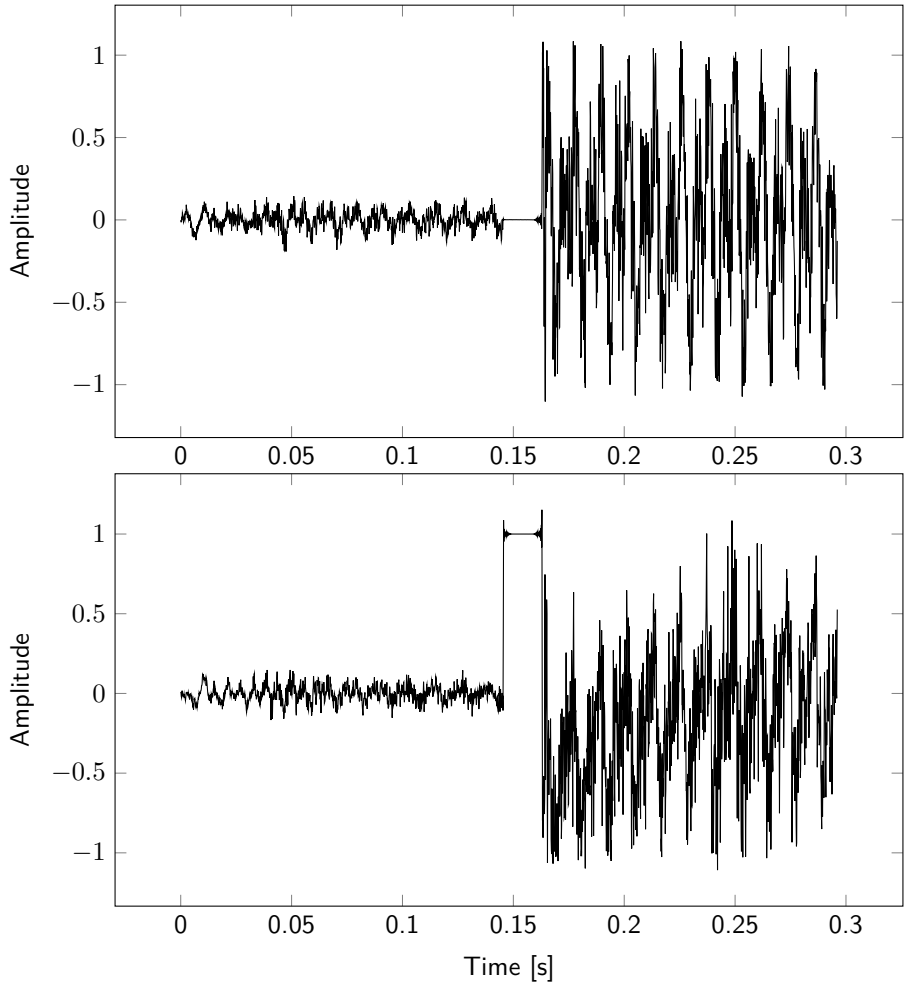


Figure 18: Effect of reconfiguration on audio signal at the receiver. The dropout lasts for approximately 1000 samples $\approx$ 20 ms

.

Heterogenous channels Remarks on single channel operation and branching above taken together, result in additional properties. One such property is duplex operation of the entire system, changing it into a two-way communication system rather than a uni-directional transmission device. Another such property is channel-heterogeneity. This means, that various different signals representing for example an audio signal and signals from other sensors such as vibration, temperature, physiological, video and so forth can be combined in the processing and transmission. If the channel capacity is sufficient, the system can be used for SDR prototyping as will be further elaborated in the next section.

4.2 Application to Software-Defined Radio

In SDR systems, often FPGAs are used as an intermediate layer between analog frontends and application software. Depending on the bandwidth the system is designed to process, some stages often need to be implemented in hardware. A rudimentary scenario is outlined in Figure 19.

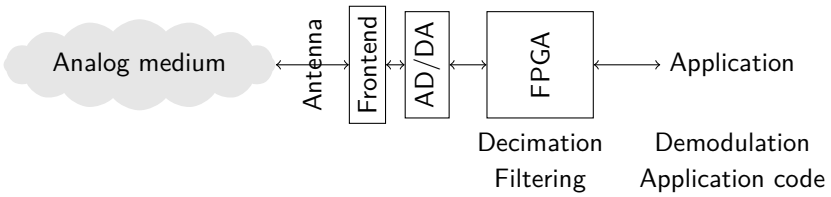


Figure 19: Simple SDR system.

In the context of the 100GET research project[16] concerned with data transmission at 100 Gbit/s an FPGA-based signal generator has been developed by the project team. Some selected problems have been identified as potentially amenable to application of DPR. Signal generation at the required rates is a challenging task and current computing hardware can only provide the data fast enough through parallelisation. Parallel data words are then serialised for transmission in a dedicated separate assembly. Looking at the system on this level, there are two main spots where PR can be applied. The first one concerns different timing adjustments in the parallel data paths in the I/O layer, the second one concerns the definition of application logic given a fixed I/O layer. This corresponds again to the HOS view. The situation is illustrated in Figure 20.

4.2.1 I/O Delays

In the setup just described, the data words are transferred to the serialiser together with a clock signal. At these speeds, the length of each conducting path needs to be considered so that all bits arrive at the destination port within a given time window. The main reasons for propagation time differences are wire lengths on the Printed Circuit Board (PCB) and the variability of routes within the FPGA over several reimplementations of the design.

Since the overall electrical lengths of the conducting paths cannot be known in advance, provision has to be made for delaying single bit lines by a specified amount of time. The bits are transmitted with a Double Data Rate (DDR)-method at 500 MHz. Assuming an ideal slew rate, the result is a 1 ns window of validity during which all bits need to be asserted. The original design already uses the IODELAY and OBUFDS primitives to realise this kind of delay. The IODELAY primitive has 64 taps with a resolution of

[16] http://www2.informatik.hu-berlin.de/sv/forschung/100get.shtml

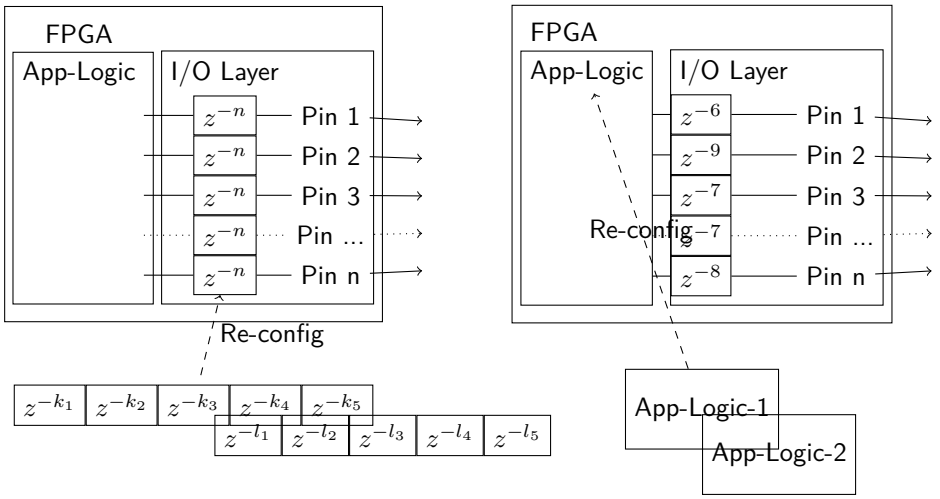


Figure 20: Principal points of application of PR in a high-speed data transmission setup.

$$T_{res} = \frac{1}{64 \cdot F_{ref} \cdot 10^6} \text{ with } F_{ref} \text{ reference clock frequency in MHz}$$

so that using an internal 200 MHz reference clock, the signal can be shifted by 78.125 ps increments [78] which enables the compensation of 2.3 cm units of length. For the 1 ns window the signal need to be shifted at most by about 12 such increments. On the Virtex-5, the IODELAY primitive acting as output delay can only be operated in the fixed mode. That means, without DPR the output delay value cannot be changed during operation. This entire procedure has to be executed only rarely during system development but still can provide a major speedup since only a small part of the systems needs to be reimplemented.

Given a simplified experimental top-level design, a RP was set up within, which transmitted the clock signal on its outputs, one with zero delay and the other with a delay value fixed per implementation of the RM. The OBUFDS primitives have to be contained within the RP. As a consequence, the top-level design needs to be told to leave these I/O pins untouched. This is done with the VHSIC Hardware Description Language (VHDL) fragment in Listing 1. The RM internal structure is displayed in Figure 21.

Listing 1: HDL fragment defining the differential output ports and surpressing OBUF generation for these ports.

```
entity PR_Clk_Delay_TL is
  port(
    Rst:          in  STD_LOGIC;
    Clk:          in  STD_LOGIC;
    D1_p_out:     out STD_LOGIC;
```

```
    D1_n_out:    out STD_LOGIC;
    D2_p_out:    out STD_LOGIC;
    D2_n_out:    out STD_LOGIC;
    LED_0:       out STD_LOGIC
    );
end PR_Clk_Delay_TL;

-- ...

  attribute buffer_type : string;
  attribute buffer_type of D1_p_out: signal is "none";
  attribute buffer_type of D1_n_out: signal is "none";
  attribute buffer_type of D2_p_out: signal is "none";
  attribute buffer_type of D2_n_out: signal is "none";
```

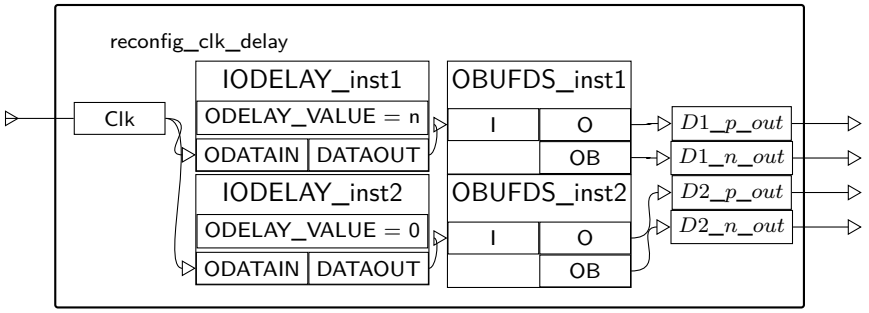


Figure 21: RM internal structure for ODELAY testing.

The result, as for example viewed on an oscilloscope, are two waveforms being shifted in phase relative to each other, as is displayed in Figure 22. The amount of the phase-shift depends on the value of n in the IODELAY instantiation. The hardware used in this experiment was an AVNET-LX50T development board, the outputs were two pairs of differential pins wired to the on-board SubMiniature Version A (SMA) connectors.

In this way, all ODELAY configurations needed can be pregenerated for different n and loaded during runtime until the signals fit, which is determined by "manual" inspection. If the system were designed to contain a feedback channel for determining the offsets, adjustment could be achieved automatically. If one limits the problem to the chip perimeter, this could readily be achieved by use of the CAPTURE_VIRTEX5 primitive together with IODELAY reconfiguration, see also [25].

4.2.2 Word delay

On a slightly different scale, there is another timing related problem in the 100GET setup. This is the synchronisation problem of four parallel data streams, which are

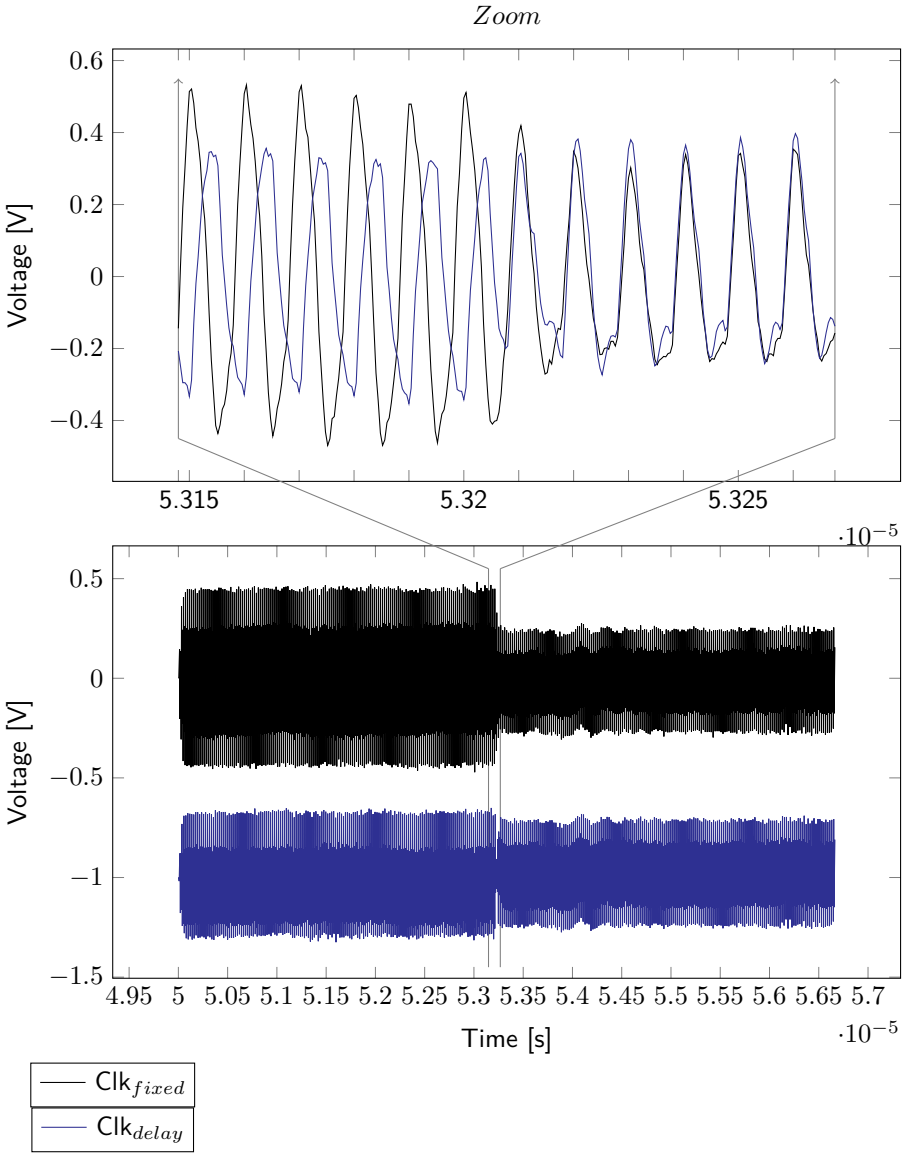


Figure 22: Clock signal behaviour during reconfiguration while modifying ODELAY attribute with PR.

emanated from the generator. This problem stems from phase-ambiguities in the derived output clock. These ambiguities result from uncontrollable initialisation conditions. Each parallel word output has to be shifted by potentially the entire word length in order to be in sync with the three remaining streams. This is achieved in principle by splitting the word into two parts at bit k. Assuming that the Least Significant Bit (LSB) leaves the serialiser first, the less significant portion of the word is left-shifted by k bits and

output immediately whereas the remaining more significant bits are buffered for one clock cycle and prepended to the left-shifted part of the next cycle in the less significant position of the next word. After serialisation this results in a temporal shift of every bit by an amount of k positions.

An existing solution is using a multiplexer to achieve dynamic displacement within an otherwise fixed structure. The input to this structure is a shift value via a dedicated control line. Applying DPR to the same problem, the structure can be internally fixed and thereby dispense with control logic. In this manner, resource usage can clearly be reduced. The simplified structure of the multiplexer and the respective RM is depicted in Figure 23.

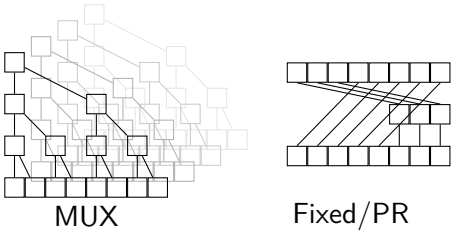


Figure 23: Spatial extension of multiplexer structure compared with a fixed delay structure that realises dynamical delays through reconfiguration.

For such a single bitwise-wordshift module with a word length of 128 bits the numbers given in Table 2 can be extracted from the device utilisation summary after synthesis.

Table 2: Device utilisation for static and dynamic wordshift circuits.

Logic	MUX	Fixed (Shift: 1)	Fixed (Shift: 127)	Available
Registers	1305	130	257	44800
LUTs	1153	129	256	44800

The fixed modules are those used with the DPR method because the circuit's I/O behaviour does not need to change for one single configuration. The actual utilisation of resources depends on the shift argument while this number does not change in the MUX case. In addition, the overall timing behaviour will improve in the fixed case because the signal in effect only traverses one buffer stage (depth is 1).

4.2.3 Reconfigurable Application

The two methods presented above for adjusting timing related characteristics of bit lanes clearly belong to the I/O layer. The parameters that are changed by these modules will settle to some arbitrary but otherwise fixed values during the overall design cycle, because

e.g. the PCBs involved are not changed anymore. The output layer together with the reconfiguration logic can be regarded as the HOS of this particular system.

The application itself can now put into the remaining free space. Application design can also be done in a reconfigurable way. This situation corresponds to the right hand part of Figure 20. Of course it does not need to be as monolithic as depicted there but can again be broken up into co-operating concurrent circuits. This is subject to general considerations of process decomposition and modularisation with respect to waveform generation and analysis. This will not be further elaborated here but some starting points are given for example in [48, 40].

4.3 Direct Bitstream Manipulation

While everything outlined above relies on recourse to conventional design concepts at various stages, direct bitstream manipulation at the same time poses substantial challenges and promises some interesting byways. These pertain to the flexibility of RMs, to the speed of creating configurations, and to the application of non-standard design procedures.

Conventional design concepts mainly consist of the manipulation of HDL code. There is much research in high-level approaches, most of which boils down to generating HDL once the high-level activity is finished. The high-level activity consists in the spatio-temporal partitioning of the application processes [10, p. 99]. Either way, the hardware description on that level is then passed through the vendor-devised tool-flow to generate a configuration bitstream. The final step in this flow is the translation of a fully routed and annotated netlist into the format required by the configuration memory. By direct bitstream manipulation, on the other hand, the entire flow is bypassed, and the configuration bitstream is modified directly. Both procedures are illustrated graphically in Figure 24. The bitstream representation is offering itself for the application of GA while at the same time a cycle of modification and configuration is accelerated profoundly. Alas, since the format of this file is not documented fully, valid modifications have to be inferred from the existing vendor documentation, prior work in independent bitstream analysis and by additional analysis of specifically generated bitstreams.

There exists some vendor-independent work on DBM with different objectives. There is for example the problem of partial bitstream compacting for decreased configuration times as realised with *combitgen* and documented in [13]. There is the problem of bitstream relocation for implemented cores which has been tackled with the *pbitpos* [34], *parbit* [29] or the *BiRF* and *BAnMaT* [16] tools. Finally, considering the reverse direction, bitstream analysis is immensely valuable for all generative methods at the bitstream level. Candidates here are *debit*, which is described in [50] and for example *FAT* (FPGA Analysis Tool) [32]. Another remarkable approach is that of Hübner et al., who realised an online on-chip resource utilisation viewer, directly accessing the VGA port from the FPGA. The main practical problem with all of these works is, that they are specific to the Virtex-2 (Pro), Virtex or earlier platforms, including JBits, and are mostly not generally available. There is no such tool that will work with more recent generations of Xilinx FPGAs, and if they do, they are limited to specific devices within the chip families as is the case with debit and FAT.

4.3.1 Bitstream format and configuration details

This section serves as a collection of information regarding the structure and layout of the Xilinx Virtex-5 configuration bitstream. Configuration data in the bitstream is divided into Configuration Frames (CFs). This is the smallest addressable unit in configuration memory and comprises of 41 32-bit words totalling in 1312 bits per CF. Reconfigurable frames are built from a discrete number of CFs. These units are derived from the row and column tiling of the device. Each column is made up of a fixed type of primitive and

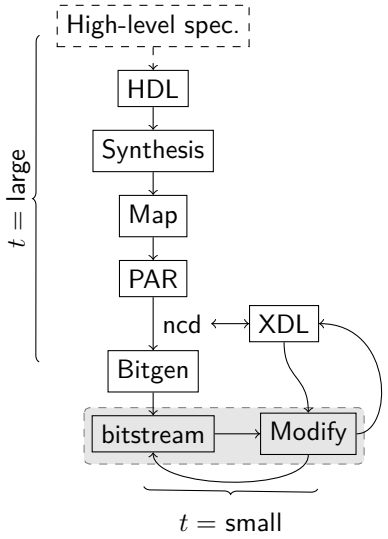


Figure 24: Different steps and processes involved in a hardware design flow. Entry is done via an HDL or possibly custom structural methods which translate to HDL later. The HDL is then passed through the three stages of synthesis, component mapping and place and route, resulting in a fully annotated netlist (ncd file). At this stage, the ncd can be translated to XDL and vice versa but any modification done in XDL still needs to go through bitgen.

the type of primitive contained within the reconfigurable frame determines the number of CFs needed for configuration. For CLBs these are 36 CFs on the Virtex-5, the numbers for different types of blocks are given in [76, p. 135]. Out of these frames, 26 are used for interconnect configuration, for some types two frames configure the interface for the block and the remaining frames are block specific configuration data such as LUT equations. An alternative detailed description of the corresponding situation on the Virtex-4 can be found in [5]. A partial exploration is given in Figures 25(a) through 25(d). Anticipating the discussion in the section on DBM, a simple configuration utilising a single 4-LUT within a single RF has been created. This configuration has seven connections in total, 5 of which are for data. The base partial bitstream has been modified with a *RapidSmith* program by writing random integers to the random words in subsets of CFs constituting the RF. The result has been visualised with debit. Another empirical investigation yields the following result: In that case, the CFs of one RF were iterated and one after another set with all bits equal to one. Analysis was done both graphically for inspecting connectivity as well as using the *lutdump* mode of debit. There are indeed four frames as expected which effect different LUT configurations although those same frames appear to also contain config-bits pertaining to connectivity, see Table 3. Note that of course debit is not a definitive source, but, according to vendor documentation, block and interconnect configuration should be contained in separate

CFs and block configuration should only start after frame #27.

Table 3: Table of connection and LUT sensitivity over CF index

Mode	CF word values	CF indices eliciting changes
graphical	0xFFFFFFFF	0,1,2,6,7,12,13,14,15,16,17,18,19, 20,21,22,23,24,25,36
lutdump	0xFFFFFFFF	0,1,2,36

Configuration Frames (CFs) are accessed via their Frame Address, which has the structure given in Table 4. The block type addresses a space of eight distinct types, four of which are used, which in turn mainly serve to separate BRAM content from normal block and interconnect configuration in the bitstream but which are also used in the partial reconfiguration process. The addressing scheme is also illustrated in Figure 26 and Figure 27.

Table 4: Frame address structure

	Unused	Block Type	Top / Bottom	Row Addr.	Major Addr.	Minor Addr
Bits	31 - 24	23-21	20	19-15	14-7	6-0

As for the format within one single CF, the Virtex-5 Configuration Guide [76, Frame Bits on p. 131] states that

> A row consists of a stack of basic blocks (20 CLBs, 40 IOBs, 4 block RAMs, etc.) with a row of HCLK tiles passing through the middle. Thus, out of the 1312 bits in a frame, 640 bits are found in the basic blocks located above the row of HCLK tiles, 640 bits are found below, and 32 bits are used inside the HCLK tiles. Of the 32 bits in the HCLK tile, the 16 MSBs are unused, the 12 LSBs are used for the ECC bits, and the 4 remaining bits are used as miscellaneous configuration bits for circuits inside the HCLK tiles.

These three sections are arranged symmetrically in a single frame, as can be deduced from the above description. The bitstream not only contains the raw configuration data but also instructions for the configuration logic. The differences between a total and a partial bitstream is exemplarily illustrated in Table 5. These differences appear to reside mainly in the operation of the configuration logic. The raw configuration data itself is contained within one WRITE command in both case, which are highlighted in Table 5. The data within is tagged with their respective Frame Addresses and expanded accordingly during the write.

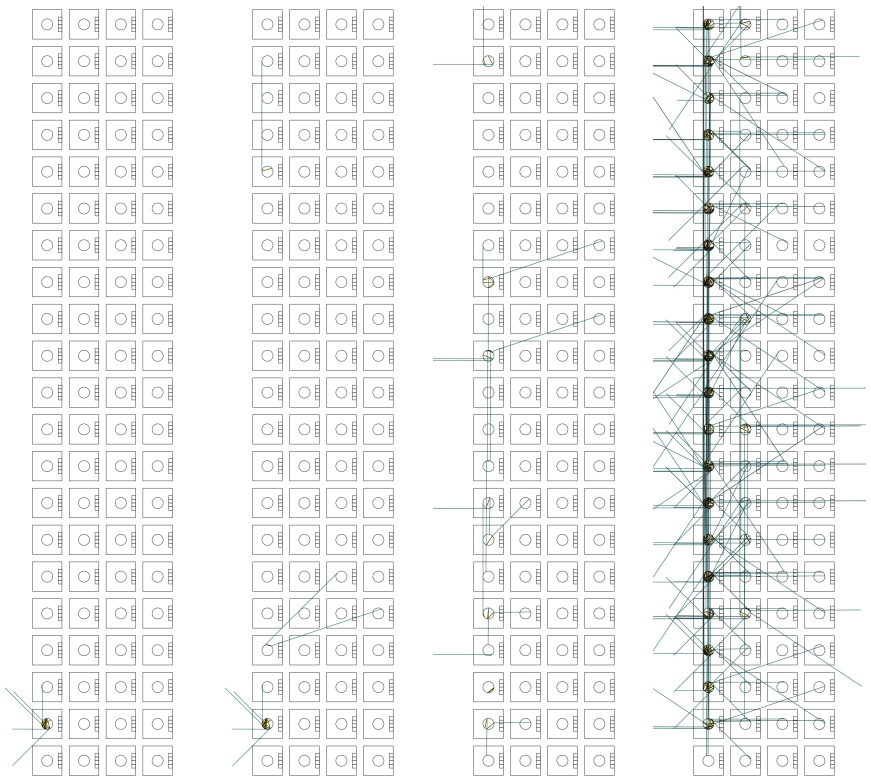


Figure 25: Different variants of randomly configured minimal RF. 25(a) renders the original, in 25(b) only the first CF has been modified, in 25(c) and 25(d) successively more randomness has been injected into the RF. debit only displays connectivity in this mode, so changes in LUT configuration and other frame are not visible here. The geometry is inverted across the horizontal axis as compared to the vendor tools. The visible region are 80 CLBs from bottom right corner of the chip, a Virtex-5 LX50 in this case. The RF is represented by the left-most CLB column.

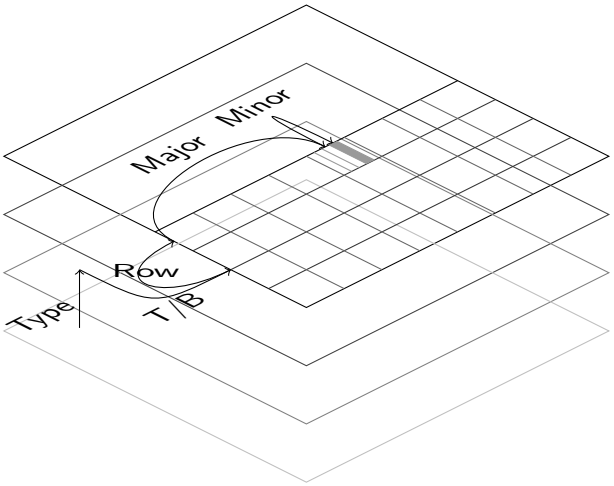


Figure 26: Geometrical interpretation of the frame address.

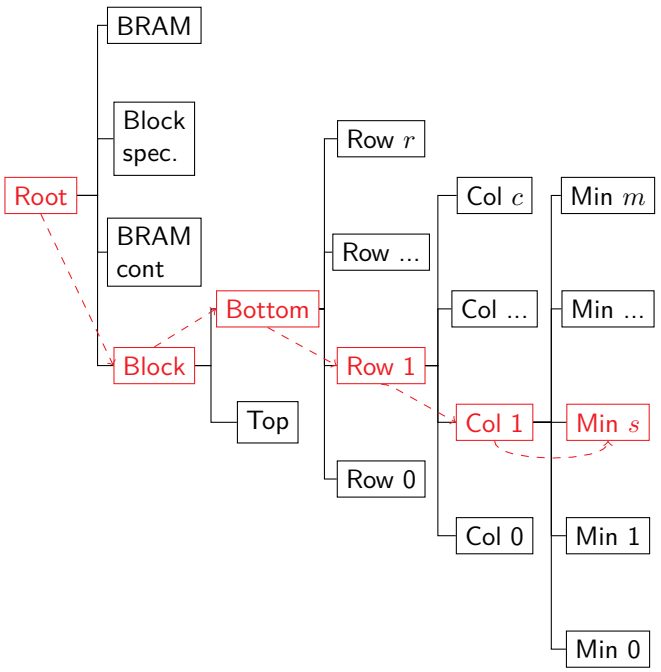


Figure 27: Tree-based interpretation of the Frame Address.

Table 5: Dissected configuration command sequence for total bitstream in the left column (l) and a partial bitstream in the right (r) column.

	Design cf_lut4_0001_routed.ncd;UserID=0:	Design cf_lut4_0001_routed.ncd;Us
	DUMMY	DUMMY
	DUMMY	DUMMY
	DUMMY	DUMMY
	DUMMY	DUMMY
	DUMMY	DUMMY
	DUMMY	DUMMY
	DUMMY	DUMMY
	DUMMY	DUMMY
	BUS WIDTH SYNC	BUS WIDTH SYNC
	BUS WIDTH DETECT	BUS WIDTH DETECT
	DUMMY	DUMMY
	DUMMY	DUMMY
	SYNC	SYNC
	NOP x 1	NOP x 1
	TYPE1 WRITE WBSTAR: 00000000 (RevisionS:	————————
	TYPE1 WRITE CMD NULL	————————
	NOP x 1	————————
	TYPE1 WRITE CMD RCRC	TYPE1 WRITE CMD RCRC
	NOP x 2	NOP x 2
	TYPE1 WRITE TIMER: 00000000 (TimerForUs	————————
	TYPE1 WRITE [UNKNOWN REG 19]: 00000000	————————
	TYPE1 WRITE COR0: 00003fe5 (CRC:Enable,n	————————
	TYPE1 WRITE COR1: 00000000 (PersistDeasD	————————
	TYPE1 WRITE IDCODE: 02896093	TYPE1 WRITE IDCODE: 02896093
	TYPE1 WRITE CMD SWITCH	————————
	NOP x 1	————————
	TYPE1 WRITE MASK: 00400000 (ICAP_sel:PrM	————————
	TYPE1 WRITE CTL0: 00400000 (ICAP_sel:Tok	————————
	TYPE1 WRITE MASK: 00000000 ()	————————
	TYPE1 WRITE CTL1: 00000000 ()	————————
	NOP x 8	————————
	TYPE1 WRITE FAR: 00000000	————————
	TYPE1 WRITE CMD WCFG	TYPE1 WRITE CMD WCFG
	NOP x 1	NOP x 1
	TYPE1 WRITE FDRI: 00000000 words	TYPE1 WRITE FAR: 00110080
Write cfg(l)	TYPE2 WRITE FDRI: 0005fbbc words	————————
	TYPE1 WRITE CRC: e85e6c55	————————
	TYPE1 WRITE CMD GRESTORE	————————
	NOP x 1	NOP x 1
Write cfg(r)	————————	TYPE1 WRITE FDRI: 000005ed words
	————————	TYPE1 WRITE MASK: 00001000 ()
	————————	TYPE1 WRITE CTL1: 00000000 ()
Routes active	TYPE1 WRITE CMD DGHIGH/LFRM	TYPE1 WRITE CMD DGHIGH/LFRM
	NOP x 100	NOP x 101
	TYPE1 WRITE CMD GRESTORE	————————
	NOP x 30	————————
	TYPE1 WRITE CMD START	————————
	NOP x 1	————————
	TYPE1 WRITE FAR: 00ef8000	TYPE1 WRITE FAR: 00ef8000
	TYPE1 WRITE MASK: 00400000 (ICAP_el:PrM	TYPE1 WRITE CRC: 563de185
	TYPE1 WRITE CTL0: 00400000 (ICAP_sel:Tok	————————
	TYPE1 WRITE CRC: 0c90449e	————————
	TYPE1 WRITE CMD DESYNCH	TYPE1 WRITE CMD DESYNCH
	NOP x 61	NOP x 1

4.3.2 Modifying BRAM content

On-Chip BRAM resources can be accessed through a small amount of logic conforming to the BRAM interface requirements from both custom hardware as well as a microprocessor through a BRAM controller component connected to the processor bus. For slowly changing BRAM contents, this logic can be saved on the processor side by reading and writing BRAM contents through the ICAP device.

Using the reference designs attached to the Xilinx UG743 PR tutorial [80], BRAM modification can be done straightaway. The design consists of a counter, that is fed into a reconfigurable module. The module contains a single BRAM cell and the counter is used as the BRAM address. Different initial BRAM content can be generated either from HDL or via the *fpga_editor*. The differing frames in the resulting bitstreams can then be searched for and their contents compared. This has been done with the *rapidSmith* framework. An example code segment is presented below together with the resulting frame differences.

Listing 2: Code fragment for writing BRAM content

```
// grab a frame, change it, and write it back
// 3146240 is FAR=00(3)00200 in hex
// a BRAM content frame: Type=2
// in the bottom half: uneven = 3
fpga.setFAR(3146240);
// iterate over first 8 frames in BRAM segment
for(int i = 0; i < 8; i++) {
    Frame fr = fpga.getFrame(fpga.getFAR());
    FrameData frda = fr.getData();
    frda.zeroData(); // zero all data in the frame
    if(i % 2 == 0)
        frda.setData(10, 1); // word index, value
    else
        frda.setData(11, 1);
    // configure frame with this frame data
    fr.configure(frda);
    fpga.incrementFAR();
}
```

This results in the frame contents which are displayed in Table 6 while the same frames in the original configuration are those of Table 7. The final example in Table 8 lists the frame contents of the same frames again, this time the memory has been initialised with an increasing sequence of numbers. As can be seen, the bits are referenced in a non-contiguous manner within the configuration memory. Frame contents are displayed with the command given in the example below.

```
$ java
edu/byu/ece/rapidSmith/bitstreamTools/examples/FrameContents \
  -i bitstream.bit | grep -v Not\ con | less
```

Table 6: BRAM CF content for modified bitstream.

FAR=00300200, bottom Type=BRAM (1), Row=0, Column=4 (BRAMCONTENT), Minor=0
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000001 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000
FAR=00300201, bottom Type=BRAM (1), Row=0, Column=4 (BRAMCONTENT), Minor=1
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000001 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000
...

Table 7: BRAM CF content for original bitstream.

FAR=00300200, bottom Type=BRAM (1), Row=0, Column=4 (BRAMCONTENT), Minor=0
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 0F00000F F00000F0 00000000 00000000 00000000 00000000
00000000 00000000 0F00000F F00000F0 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000
FAR=00300201, bottom Type=BRAM (1), Row=0, Column=4 (BRAMCONTENT), Minor=1
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 0F00000F F00000F0 00000000 00000000 00000000 00000000
00000000 00000000 0F00000F F00000F0 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000
...

Table 8: BRAM CF content for increasing number sequence.

FAR=00300200, bottom Type=BRAM (1), Row=0, Column=4 (BRAMCONTENT), Minor=0
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000001 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000
FAR=00300201, bottom Type=BRAM (1), Row=0, Column=4 (BRAMCONTENT), Minor=1
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00010000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000
FAR=00300202, bottom Type=BRAM (1), Row=0, Column=4 (BRAMCONTENT), Minor=2
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000001 00000000 00000000 00000000 00000000 00010000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000
FAR=00300203, bottom Type=BRAM (1), Row=0, Column=4 (BRAMCONTENT), Minor=3
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00010000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000
...

4.3.3 Genetic Algorithm for intrinsic evolution of LUT configuration

Direct bitstream manipulation opens interesting pathways for applying machine learning methods to circuit design. As discussed in the introduction, two important experiments have been conducted by Thompson in 1996 [69] and later by Bird and Layzell in about 2002 [8] that outline the principle of applying Evolutionary Algorithms (EAs) to the configuration of circuits, in Thompson's case that of FPGAs. Although these experiments feature fully *unconstrained* evolution, this path has not been chosen here for reasons of simplicity. The difficulty lies in preventing potentially hazardous configurations to be written to the FPGA during exploration of the configuration space, although the actual danger of partially or completely destroying the chip seems to be lower than expected initially[17]. For a general introduction to EAs see Appendix C.

Intrinsic evolutionary circuit design is possible without PR, see Figure 28. Intrinsic in this context means, that the evaluation of individuals is done "in silico" and not in simulation. The motiviation behind using intrinsic evaluation is at least two-fold. For one, it is the possible exploitation of circuit characteristics that are lost by the simplifying assumptions of the models underlying simulations. Yet more importantly though, intrinsic evaluation can result in major speed-ups compared to evaluation in simulation.

The use of PR techniques in the evolutionary context opens up the possibility of building a completely autonomously evolvable System-on-Chip. This requires that the EA can be executed entirely on components contained within a single chip. In the present case, this can comfortably be realised on the embedded Linux system using existing libraries, although many implementations of EA in hardware can as well be found in the literature, some examples are [47, 82, 72, 18]. The target of evolution in this case are circuit fragments which reside in Reconfigurable Partitions (RPs). A memory efficient implementation for example is the Compact GA that encodes not the bits themselves but only the probabilities of these bit becoming either '0' or '1'.

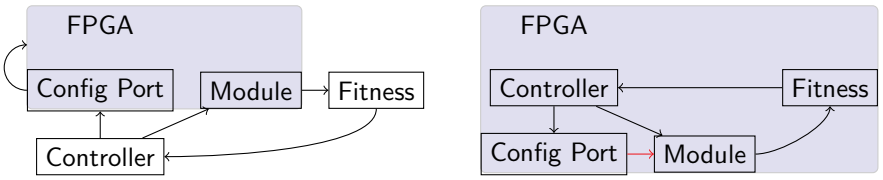


Figure 28: Evolvable system with and without DPR.

The controlling system in that case has to be able to generate new bitstreams. Since vendor tools are not available for either the Microblaze or PPC platforms, this route of bitstream generation is barred for autonomous on-chip systems. The ability of loading

[17] http://forums.xilinx.com/t5/Spartan-Family-FPGAs/ICAP-destroy-FPGA/td-p/62470

a configuration, once generated, onto the chip is provided by the Linux ICAP interface.

This experiment is about the simplest arrangement by which the viability of this approach on current hardware could be demonstrated. The goal was to evolve a configuration realising an arbitrary but fixed four-input logic function. This function can be fully specified by a binary vector of length 2^4, which represents the rightmost column of the logic function's truth table. An example is given below.

Table 9: Truth table for a generic four-input Boolean function. Allocation of the y_n defines one of 2^{16} possible functions.

x_1	x_2	x_3	x_4	y
0	0	0	0	y_1
0	0	0	1	y_2
0	0	1	0	y_3
0	0	1	1	y_4
0	1	0	0	y_5
0	1	0	1	y_6
0	1	1	0	y_7
0	1	1	1	y_8
1	0	0	0	y_9
1	0	0	1	y_{10}
1	0	1	0	y_{11}
1	0	1	1	y_{12}
1	1	0	0	y_{13}
1	1	0	1	y_{14}
1	1	1	0	y_{15}
1	1	1	1	y_{16}

A configuration for the four-input AND-function would be

$$y = (0, 0, 0, ..., 0, 1)$$

out of 2^{16} possible configurations. All possible binary vectors of 16 elements represent the search space for the algorithm. Evaluation consists of applying all 16 possible inputs to the circuit and recording each answer. Fitness is then calculated as the Hamming distance between recorded vector and target vector which obviously must be zero

$$f_{fit} = 2^4 - d_{ham}(y_{out}, y_{target})$$

The evolutionary framework[18] was tested in simulation first. In this case, the genotype equals the phenotype and the EA's task amounts to finding the correct bit-vector out of the $n = 2^{16}$ possible vectors. A purely random search without repeating values

[18] http://pyevolve.sourceforge.net/

takes $\frac{n+1}{2}$ steps on average. The GA finds the solution in about 20 generations with a population size of 10 and a vector length of 16, see an exemplary run in Figure 31. The search was also conducted for a vector of size 128 where the GA finds the solution in about 300 generations with a population size of 20 and a slightly lower mutation rate of $p_{mut} = 0.01$. An exemplary run can be seen in Figure 32. The algorithm used is a simple GA without elitism.

The same experiment was repeated with intrinsic evaluation of the individuals. The tool used for bitstream manipulation was RapidSmith[19]. There is an alternative to RapidSmith called Torc[20] which is approximately functionally equivalent. The difference between these two is their implementation language. The choice of tool was due to the respective project's status at the time of the experiment. Since RapidSmith is written in Java, the bitstream parsing and modification was done on a standard workstation computer in order to circumvent the installation of the Java environment on the embedded PPC Linux system. Using the C++-based Torc this would be expected to leave a smaller footprint on that platform. For the principal demonstration of feasibility of the method this setup was deemed sufficient.

The prerequisites are as follows. The RP is populated by a module adhering to the interface given in Listing 3 below.

Listing 3: HDL fragment of the four-input logic function block.

```
entity fourlut is
    Port ( I4 : in  std_logic_vector (3 downto 0);
           O1 : out  std_logic);
end fourlut;
```

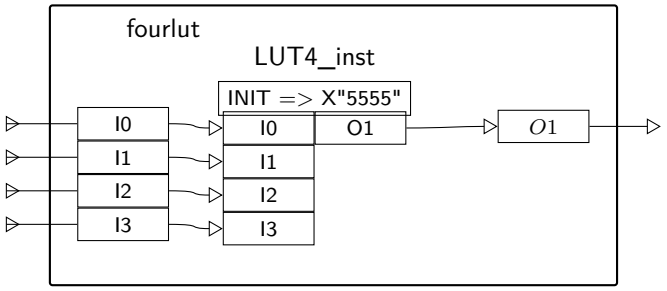


Figure 29: Simple 4-LUT module implementation.

The module contains a single LUT4 primitive instance which directly implements the four-input logic function, as can be seen again in Figure 29 and in Figure 30. By generating several variations of the LUT configuration, the relevant bits in the bitstream were isolated. These locations were coded into the genotype to phenotype transformation implemented in a RapidSmith module and are given in Listing 4 below. This

[19] http://rapidsmith.sourceforge.net/
[20] http://torc-isi.sourceforge.net/

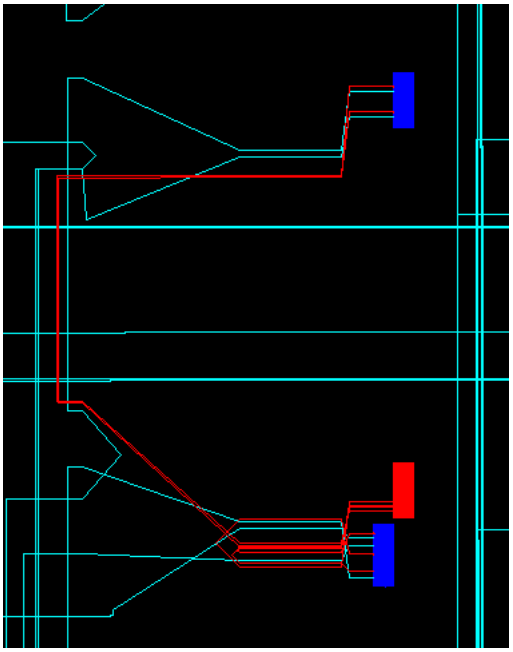


Figure 30: Simple 4-LUT module in fpga_editor view.

is valid only for the specific module implementation using a LUT configured in frame $0x0001131A$ and its three immediate successors. It is indeed not clear why the configuration information for a single LUT is spread redundantly over four consecutive CFs, as is exemplarily displayed by the bitstream difference operation of Listing 5 whose output is given in Table 10.

Listing 4: Code fragment indicating bit positions for specific LUT configuration bits within the CF

```
// setup bit positions
int[] bitpos03 = {688, 696, 692, 700, 690, 698, 694, 702, \
                  689, 697, 693, 701, 691, 699, 695, 703};
int[] bitpos12 = {689, 697, 693, 701, 691, 699, 695, 703, \
                  688, 696, 692, 700, 690, 698, 694, 702};
```

Listing 5: Command for generating bitstream differences.

```
$ java BitstreamDiff -d -i id_lut0400_routed.bit \
  -c id_lut4000_routed.bit
```

The software bitstream modifier is called with four parameters. These are a bitstream template file, the current logic function to be implemented, which is supplied by the GA and which is fixed for one entire evolutionary run, the CF index where the particular LUT resides (0x0001131A in this case) and an output bitstream filename. The module loads the bitstream template, extracts the specified CF, changes the bits as specified by the

current genotype and writes the bitstream containing the modified CF back into a new file. This is followed by three more steps. The first is copying the current bitstream over to the embedded system, triggering hardware reconfiguration by writing the bitstream to the ICAP device and finally executing a custom program which writes all possible inputs to the RM and records its output. The program returns a string of 16 bits corresponding to the hardware module's response to each input, see Listing 6 below. From this, the Hamming distance to the target bitstream and hence the individual's fitness can be calculated and the GA can proceed to the next evaluation. As expected, the GA finds a solution by utilising intrinsic evaluation after about the same number of iterations as in the simulation. This is demonstrated by the graph in Figure 33.

Listing 6: Fragment from evo_binfunc.c for fitness evaluation of the hardware module.

```
// loop over possible input values (0000, 0001, ..., 1111)
for(i = 0; i<16; i++) {
  // bf_in is a pointer to the GPIO output register
  // connected to the module
  *(bf_in) = i;
  usleep(10); // wait
  printf("%d", *(bf_out));
}
printf("\n");
```

Timing A note about the temporal requirements of reconfiguration and those of the GA is in order in the context of this experiment, although the problem is of course much too crude for the real advantages of intrinsic evolution to peel out. A minimal RF on the Virtex-5 of 20 CLBs results in a bitstream length of 6632 bytes. The SelectMAP and ICAP configuration interfaces can achieve a bandwith of 3.2Gbps at the maximum clock rate [79, p. 100]. This yields a theoretical configuration time of about 16us. For a problem of the scale in the 4-LUT example above, evaluation time is negligible as well. Bitstream modification takes about 1s using FrameModderLogic within the RapidSmith framework. An overall run of 300 evaluations (30 generations with 10 individuals each) takes on the order of 15 minutes, the majority of which is spent in remote shell access and transfer to the embedded Linux system. Also this does not consider any options for parallel evaluation.

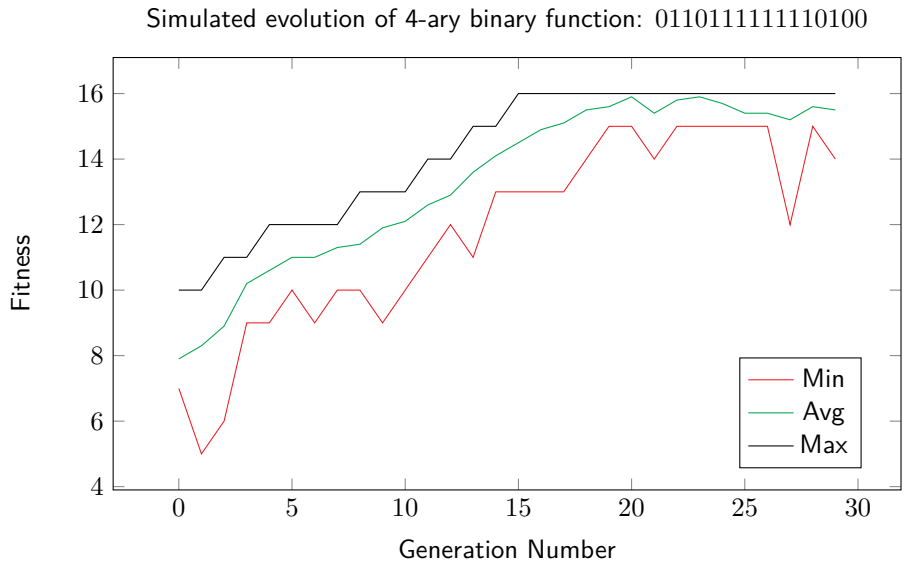
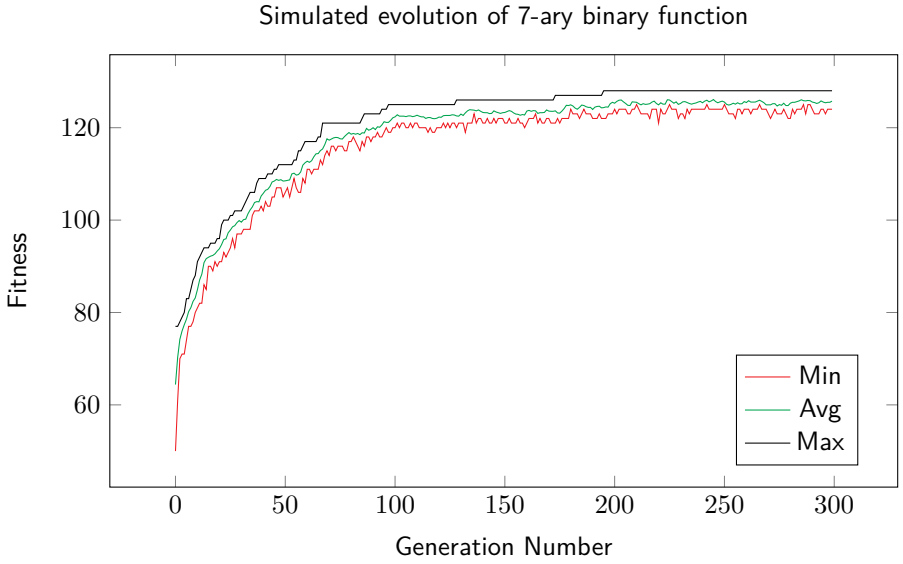


Figure 31: Evolution in simulation of 4-ary binary function.



Truth table config, split for legibility

11000101101011111100001000111100 00001000101111110000111110101110
01001100000100000100010111101101 10011111001101001101000001110011

Figure 32: Evolution in simulation of 7-ary binary function.

Table 10: Two different 4-LUT initialisations (0400 and 4000) as reflected in the CF.

0001131A (FAR=0001131A, top Type=LOGIC (0), Row=2, Column=38 (CLB), Minor=26):
Frames differ in contents
FPGA 1
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 0000058A 00000400 00000000 00000F0F
00000F0F 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000
FPGA 2
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000D88 00000100 00000000 00000F0F
00000F0F 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000
0001131B (FAR=0001131B, top Type=LOGIC (0), Row=2, Column=38 (CLB), Minor=27):
Frames differ in contents
FPGA 1
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000D8B 00000800 00000000 00000F0F
00000F0F 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000
FPGA 2
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000589 00000200 00000000 00000F0F
00000F0F 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000
0001131C (FAR=0001131C, top Type=LOGIC (0), Row=2, Column=38 (CLB), Minor=28):
Frames differ in contents
FPGA 1
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000D8B 00000800 00000000 00000F0F
00000F0F 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000
FPGA 2
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000589 00000200 00000000 00000F0F
00000F0F 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000
0001131D (FAR=0001131D, top Type=LOGIC (0), Row=2, Column=38 (CLB), Minor=29):
Frames differ in contents
FPGA 1
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 0000058A 00000400 00000000 00000F0F
00000F0F 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000
FPGA 2
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000D88 00000100 00000000 00000F0F
00000F0F 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000
...

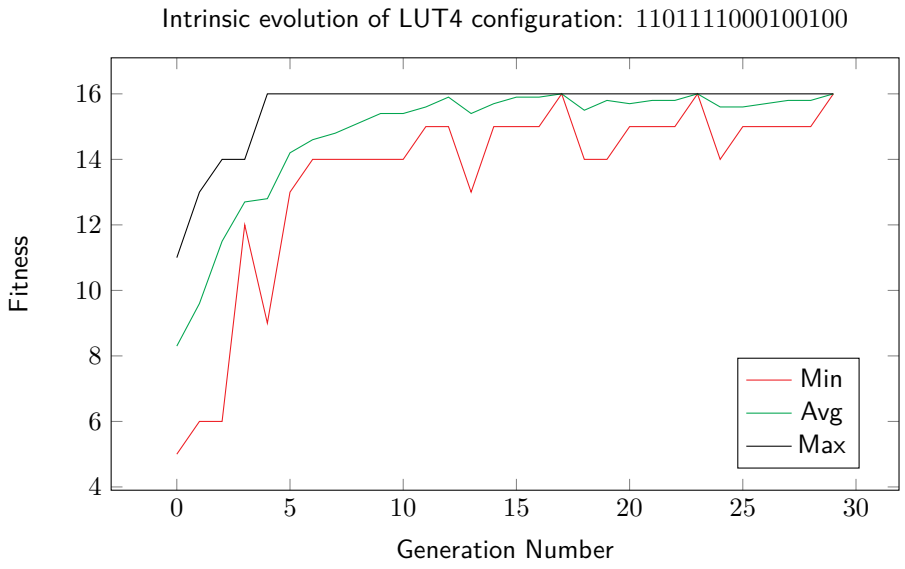


Figure 33: Intrinsic evolution of 4-ary binary function operating directly on LUT configuration in the bitstream.

4.3.4 Ring oscillator

A third and final experimental setup in continuation of LUT configuration consists of a ring oscillator confined to a RP. The ring oscillator is made up of an odd number of single bit inverters. The oscillator's frequency then is determined by the length of the chain of inverters. More concisely, it is determined by the inverter propagation delay times plus the transmission times on the lines connecting the gates. According to the Virtex-5 FPGA Data Sheet [78], the LUT switching time is only about 0.1ns so the major part of the delay seems to originate from the connecting wires and interconnect points. The smallest loop that actually oscillated in the experiments was made of five inverters with a period of 3.1ns.

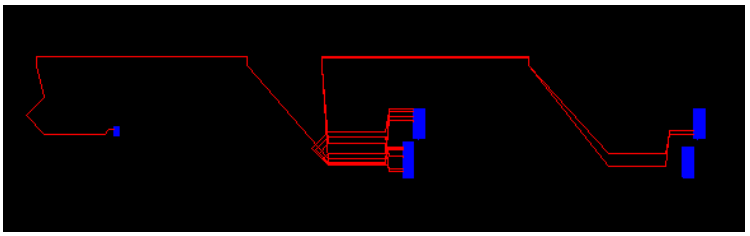
Such a structure is interesting in several regards. Since it operates asynchronously and the aforementioned delays, which add up to the overall period of oscillation, are subject to slight temperature-dependent variations, it can be used to locally measure temperature on the chip [59, 12, 64]. Also, it could be used as a local clock generator in a Globally Asynchronous Locally Synchronous (GALS) system, see e.g. [44].

Another consideration was using the initial phase of the oscillator as a random bit in a physical Random Number Generator (RNG). This was based on the idea that the initial phase "decision" of the oscillator depended on random electrical fluctuation which would push the output gate randomly into one of its two states. DPR could be leveraged in that case for continuously retriggering this initial state. Preliminary attempts at accessing the relevant information have proven unfruitful though. The tricky part is getting hold of the initial phase condition from within the Linux system after reconfiguration of the oscillator region. In addition, probing the output with an oscilloscope showed that initial phase state was deterministic. In addition, the bit rate of such an RNG needs to be considered and is approximately equal to the maximum reconfiguration rate times a parallelisation constant.

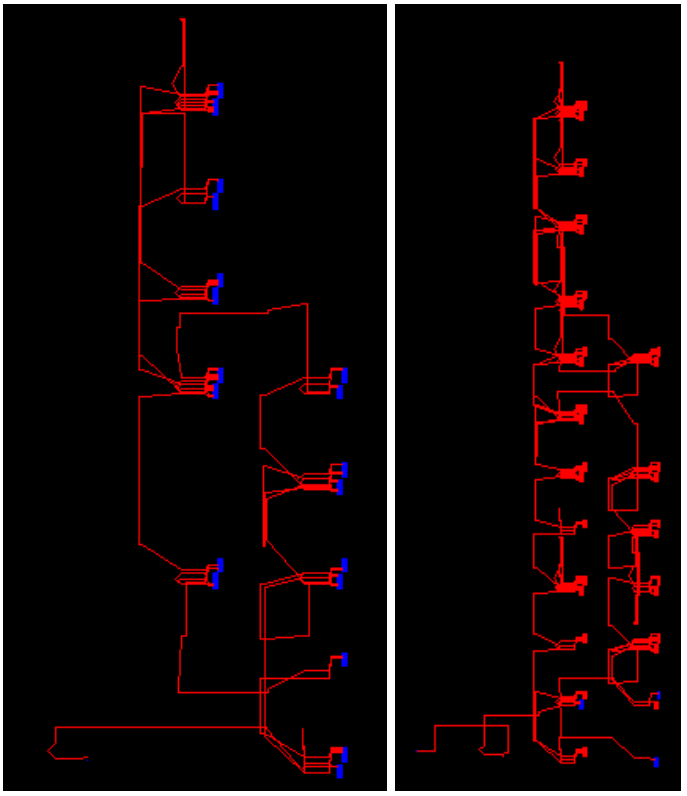
The original goal of the ring oscillator setup was to apply again an evolutionary mechanism to the netlist by extending the procedure of LUT configuration to the manipulation of signal routes. In other words, it is a preparatory step for tackling structural issues with the GA. By modifying the number of cyclically connected inverters, the EA could for example be used to find a configuration that oscillates at a specified frequency. Frequency measurement would have to be done in a separate hardware module. The additional effort went beyond the scope of this work, and so was left as is. Thus, only some results from closer inspection of the reconfiguration process are presented. First, about 20 different configurations, three of which are displayed in 34(a), 34(b) and 34(c) were investigated with regard to their resultant frequency. The outcome is plotted in Figure 35. Next, oscillator behaviour after physical instantiation has been observed. These results are plotted in 36(a), 36(b), 36(c), 36(d) and 36(e). The characteristics are deterministic per configuration over all instantiations observed, and, in particular they are independent of prior configurations. The last of the above figures features a remarkably long transient. For illustrative purposes 37(a), 37(b) and 37(c) are included. These capture the pin behaviour over an entire reconfiguration period. In the beginning of the graphs, the prior configuration is still active. This is interrupted by a deactivation

during reconfiguration which lasts for about 14us. Then, the current circuit is activated and starts to oscillate. In 37(c) it can be observed, that a ring configuration of three LUTs does not lead to oscillation. All inverter numbers have to be increased by one, as this component is included in every configuration and includes an additional enable port. The oscillator's output has been routed both to the processor and directly to an output pin.

Timing Another note on timing and numbers is apt at this point. The bitstream size for the oscillator partition is 21396 bytes, resulting in a configuration time via ICAP of 53.5us. Due to decoupling via the ICAP driver within the embedded Linux system, reconfiguration is even faster as seen by reconfiguration software, that is, reconfiguration is still going on when the write to the icap device returns. Special provision has to be taken to consider the state of the configuration logic state register (STAT) when more precision is required in reconfiguration timing.



(a)



(b) (c)

Figure 34: Three implementations of ring oscillators. 5 inverters in 34(a), 31 inverters in 34(b) and 79 inverters in 34(c)

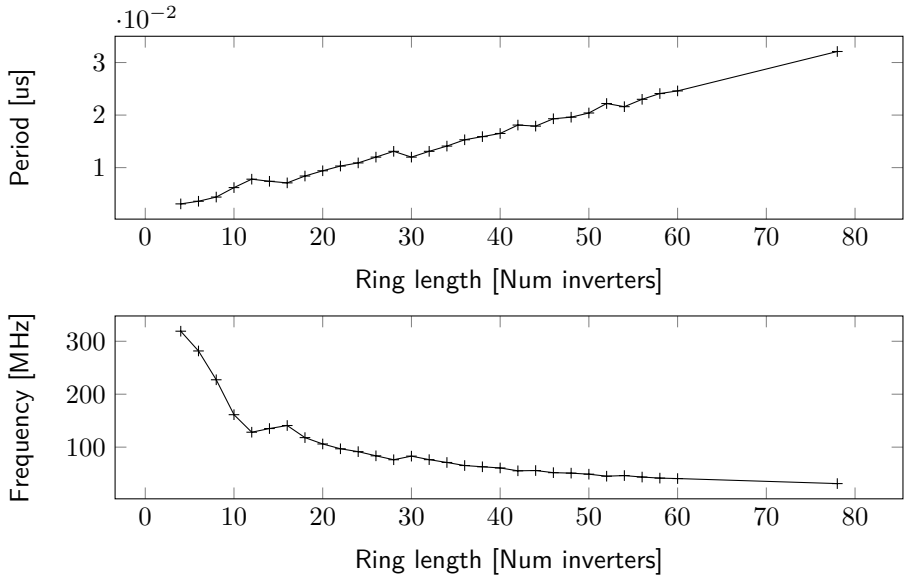
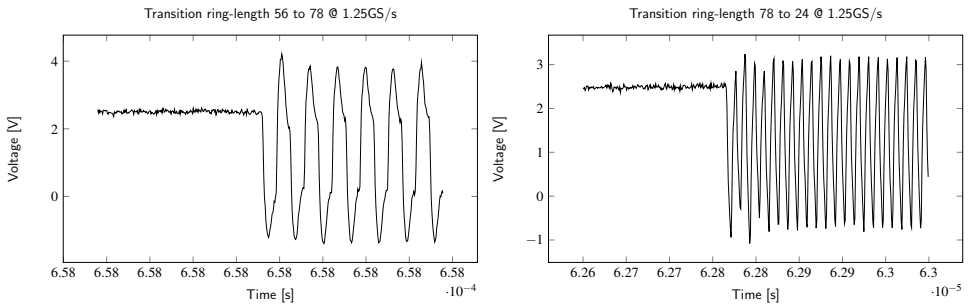
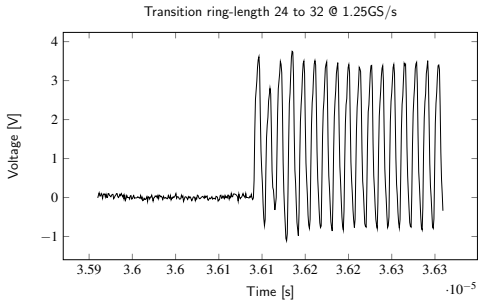


Figure 35: Oscillator frequency and period over ring length.

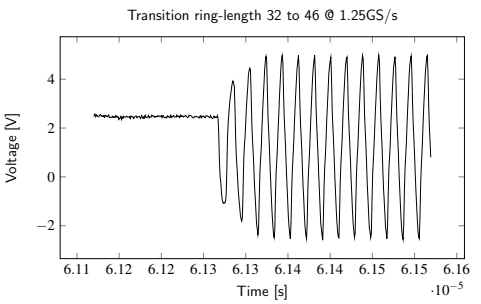


(a)

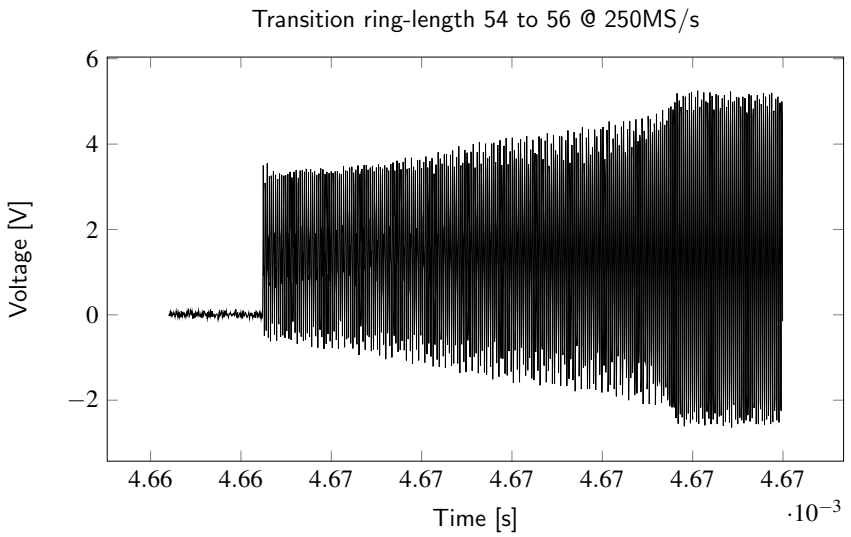
(b)



(c)



(d)



(e)

Figure 36: Transients of oscillator raw output after reconfiguration for five different configuration changes. 36(a) - 36(d) represent the more common behaviour while 36(e) exhibits a more dramatic one.

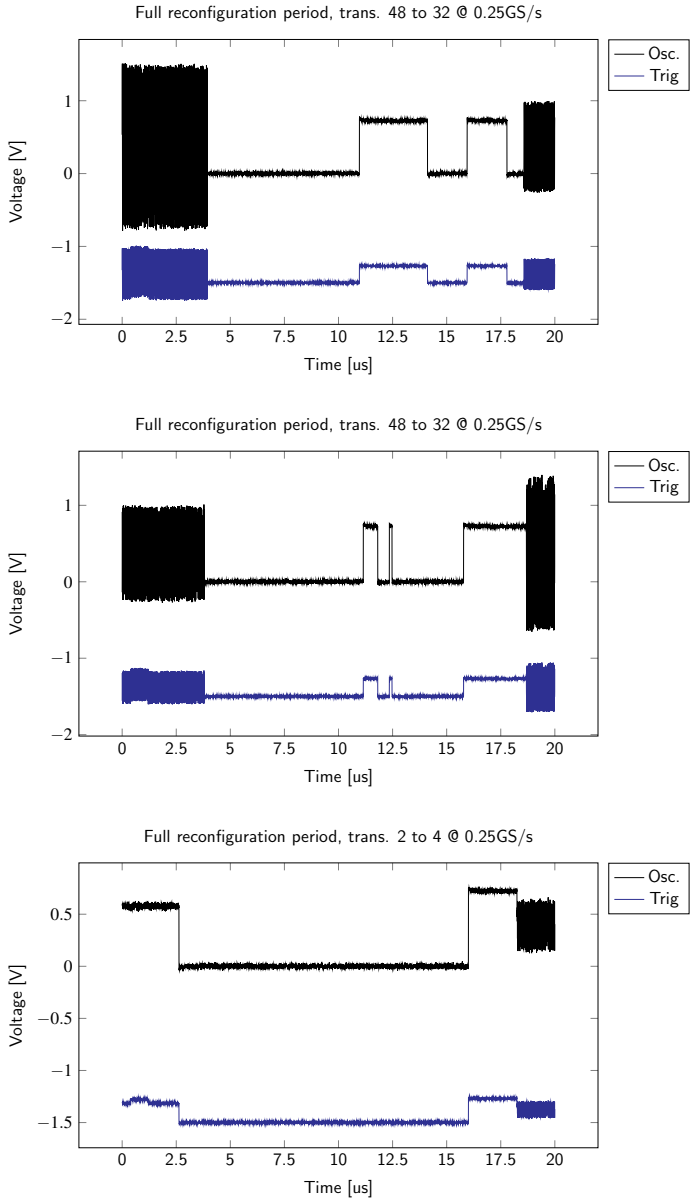


Figure 37: Entire reconfiguration period of approximately 14us for three different reconfigurations. In 37(c) it can be seen that the ring length 2 configuration does not oscillate. The trigger signal has been shifted on the y-axis by -1 for better readability.

5 Summary and perspectives

5.1 Summary

Partial dynamic reconfigurability is a key feature of current FPGA technology but taking full advantage of the possiblities remains a challenge. This is due to diverse reasons. There are the *mechanics* of reconfiguration which depend largely on the configuration technology itself and are, of course, provided by the chip vendors. Maximum reconfiguration speed for example is a direct result of the mechanics. Considerations in DPR designs reemphasize the physicality of the chip, at least in the spatial dimension, which can otherwise be neglected in many cases. The non-uniqueness of process decomposition adds an additional dimension to space over time issues already present in conventional design. One main variable in such a decomposition is the choice of granularity. Finally, reconfiguration needs to be managed in the online system which raises the problems of who is in control of reconfiguration and when and how are the required configurations generated or retrieved from a store, respectively. Finally, configuration bitstreams need to be verified to fit the currently running system.

Seen positively, there indeed exists a manageable and robust design flow as well as promising third-party alternatives. The mechanics seem to work robustly over a wide range of scenarios and can in fact handle situations beyond those reachable via high-level design. Linux and the underlying microprocessor setups already provide a solid basis for many system functionalities and this can only be expected to improve with future platforms.

In the opening sections, the topic of this work has been illuminated from various angles. The concept of hardware has been introduced as a physical arrangement capable of implementing computing processes. The differences between hard- and software with regard to their run-time modifiability and their respective modes of operation have been examined. The scope of hardware reconfigurability has been set to include a wide range of physical arrangements, transcending the domain of electronical computing in order to provide a broad embedding of these types of problems. Regarding practical aspects and current implementation platforms, a short history of reconfigurable electronic computing before and after the introduction of PLDs has been given. Finally, the application of PLD technology has been put into the context of signal processing by both general considerations and by discussing a set of rather concrete example scenarios.

This extensive introduction was followed by presenting a practical design of an embedded Linux system running on a reconfigurable platform, the ML507 Virtex-5 development board. The mechanics of Dynamic Partial Reconfiguration in this particular setting were elucidated from both the hard- and software perspective. The standard design flow for a partially reconfigurable Microprocessor system was explored and put to use. Finally, some pointers were given toward deployment on a broader range of platforms.

This set the stage for describing several experiments conducted on the system devised earlier. The intention was to demonstrate practical consequences and possibilities for hardware design and to investigate modes of operation going beyond the standard design flow. The first group of experiments retargeted an existing FPGA system for

conditional transmission of Audio-over-IP based on intrinsic signal properties. The retargeting consisted of first isolating specific hardware modules involved in the transmission and processing chain, and consequently implementing them in a reconfigurable manner. Reconfiguration was controlled by the embedded Linux system. Thereby it has been shown how the performance and compactness of a single-chip platform can be used in smart sensor nodes while achieving flexibility in prototyping and uninterrupted service in deployment. The coarse granularity of this approach posed little design overhead and benefits immediately from independent modular design approaches.

The second group of experiments dealt with specific timing and design issues arising within a larger high-speed communication system. Reduced resource consumption in a large design was explored through the use of the dynamic reconfiguration mechanism in favour of custom dynamisation logic. A special case was demonstrated where DPR could be employed to dynamically alter parts of the system behaviour in a way that was not possible in a system with static configuration. Pin behaviour of specialised I/O resources during reconfiguration was observed and could be shown to allow for glitch-free signal adjustments.

Finally, the third group of experiments was centered on the exploratory creation and manipulation of FPGA configuration bitstreams. First, details about the substratum of all design approaches, the configuration bitstream, were prepared. Then the purposeful manipulation of the bitstream was demonstrated with a goal-directed closed loop LUT modification process on the FPGA. The reduction in the time needed for generation of new configurations using this approach supports the use of "autonomous" machine-aided design methods. The ability to quickly reconfigure LUT contents moves an entire class of designs (fixed routing) within reach of evolutionary methods and modifying connectivity at the bitstream level is not expected to yield unmountable obstacles. By using asynchronous structures, low-level physical properties of circuits were investigated regarding their potential computational function and could be controlled dynamically from within the system with little overhead.

5.2 Perspectives

The perspectives acquired in the course of this work are rather broad. It is evident that reconfigurability can be included advantageously as a generic capability in systems architecture but how exactly this is to be done can only be said on a case by case basis. Basically, the challenges become more intricate in inverse proportion to the chosen granularity.

5.2.1 Toolset ecosystem

The toolset ecosystem clearly is in need of diversification. At the moment, there is a huge gap in many design flows which is populated solely by vendor tools. This means it is indeed possible to leverage alternative high-level design approaches, but, at some point the results from these stages are forced through the bottleneck of HDL to bitstream conversion. Of course there is a clash of interest between the production oriented camp

and more research oriented approaches. Reliability is of utmost importance in the first case, whereas flexibility and manipulative access move to the front in the latter case. Fortunately, recently the toolset has been enriched by the projects already referenced earlier of Brigham Young University (RapidSmith) and the University of Southern California (Torc). Both of these tools are engineered for comprehensiveness which seems to have been achieved. They have not gone the last mile though, and by this the explicit mapping of CF content is referred to. In this area, the capabilites of debit or FAT need to be polished and possibly integrated into the frameworks just mentioned. This niche needs to be filled both for synthetic as well as analytic purposes. In addition, the tools mentioned in the section on DBM above promise to provide much needed functionality, so these need to be adapted for newer reconfigurable platforms.

5.2.2 Communication and interfaces

In reconfigurable systems of any granularity, there is a need for interface conventions. These interfaces need to handle several different types of communication. On the one hand, these are high-throughput data streams such as audio-, image-, or radio data. These data need to be passed with low overhead among different processing modules. In addition, some modules need to be controlled by either other hardware modules or by a process residing within the MP system. These control interfaces are loosely characterised by a more sporadic type of traffic. The interfaces need to consider both point-to-point as well as many-to-many communication links. When the system becomes larger and/or the granularity smaller, the communication paradigm employed needs to be scalable, whereby the Network-on-Chip (NoC) approach could be utilised. Generally, these problems need to be abstracted for automated generation of the necessary substructures.

5.2.3 Microprocessor Operating System (OS)

Linux serves well as a versatile and powerful development environment but there are cases where either this part of the systems can be made lighter or where timing needs to be controlled more tightly. Especially when interacting with rigidly timed circuit instances, the scheduling jitter can be an obstacle. This calls for the use of a real-time Linux variant or, alternatively, some other Real-Time Operating System (RTOS) altogether. In addition, it needs to be seen, whether classical hot-plug concepts in OSs can be made subservient to reconfigurable architectures or if this layer needs to be extended.

5.2.4 Coarse granular generalisation

As already indicated in the relevant sections above, it would be desirable to develop a general coarsely grained framework for a partially reconfigurable stream processing system. This would require adaptable stream formats and rates, flexible routing among components and a fixed specification of the HOS layer which performs the physical I/O functions. Such a framework could be used for prototyping of generic waveform generation and analysis systems and be operated in real-time in a way similar to existing

block-based signal processing frameworks such as gnuradio/grc[21], puredata[22] or other, similar ones. There are some strongly related problems. One of these is that VHDL does not inherently support the reconfiguration concept, that is, there are no dynamic elements supported by the language directly so these need to be inserted into the design flow on top of the HDL. One approach at an HDL for reconfigurable systems is JHDL [6]. Another problem is that of resource virtualisation, which is a common feature of OSs. This then leads to the need for solving relocatability of modules in combination with temporal placement.

5.2.5 Low-level representation, introspection, and fault tolerance

The low-level approach is intertwined with a number of related issues. One of these is introspection. This is a necessary ingredient for any kind of adaptive and purposefully self-modifying system. There are several possible modes of introspection for an SoC. That of explicit observation of modules by other modules and that of register capture via hardware primitives combined with readback. These methods relate to dynamic attributes of the system but also structural introspection is very desirable of course and this can be gained through wholesale configuration memory readback. This could in principle be taken advantage of to let a system self-modify whose high-level design is not even known a priori.

For the case of testing a given system in terms of its tolerance to failures, DBM provides an efficient way of simulating SEUs in the configuration memory. Random errors can be injected into a partial bitstream with the desired distribution and the behaviour of the system can be evaluated directly after reconfiguration of the device.

The readback mechanism also plays a vital role in continous partial scrubbing, which has been mentioned further up already. Scrubbing is something like "best current practice" in real deployed systems and this practice can be considerably enhanced by doing it partially. If it is done blindly, the only advantage is increased overall system uptime. But using introspective measures, scrubbing can be done on-demand. How system failures can be detected from the modes of introspection enumerated above depends on the overall system architecture.

Here some examples need to be pulled in for further illustration. There exists abundant material in the literature about making use of biologically inspired methods to design for fault tolerance. An example is Mange's Embryonics [41] project, another one is exploiting evolutionary techniques for intrinsically fault tolerant design [71]. In the latter case, existing functions are re-optimised through adaptive methods for additional objectives while retaining fitness at other levels of functionality. This procedure can of course be applied to other objectives, such spatial extension, energy consumption etc. These approaches mostly are population based, which can be regarded as a generalisation of straight redundancy as in e.g. TMR.

Another example, which offers itself for population based computation as well as for

[21] http://gnuradio.org/, http://www.joshknows.com/grc
[22] http://puredata.info

application of the direct LUT manipulation outlined above, is that of an entirely LUT based architecture with a fixed routing. Such an approach is devised and described in Haddow and Tufte [24, 70]. An additional aspect touched upon by such a fine grained setup is the inter-module communication problem. Communication is realised in a local manner whereby only neighbouring modules are able to communicate.

But finally, all adaptive and machine-aided design and modification methods are not worth a lot, if the solutions found thereby cannot be analysed. This has been an issue for researchers since the inception of the field of e.g. Evolvable Hardware (EHW) but has not been solved comprehensively so far. Either the solutions were particular to specific devices or refuge has been taken to virtual (intermediate) FPGA models which are under full experimental control and only later mapped onto the effective technology. This represents an important instrumental gap and points back to issues of introspection, configuration readback and constructive decomposition of CFs content.

5.3 Acknowledgements

I would like to thank all those, who have supported me in different ways during the preparation of this work. These are the Signal Processing and Pattern Recognition group at the Department of Computer Science of the Humboldt-Universität zu Berlin, headed by Beate Meffert, in particular Frank Winkler, Markus Appel, and Martin Brückner, for their very generous and knowledgeable support. I am indebted to those who have additionally commented on the work and text which were Christian Blum, Jonathan Kemp and Matthias Kubisch. In addition I have enjoyed inspiring discussions with Marco Zemke, "the Killats" $(1-p)$, members of the Cognitive Robotics group, the Neurorobotics Research Laboratory and the Seminar for Media Studies (all of the Humboldt-Universität zu Berlin) as well as members of the xxxxx micro_research project, among many others. All of this before the background of my family's unconditional support.

Appendices

A Tools

This appendix serves as a quick summary of the tools used in this work as they are an important ingredient in system setup and inspection. There are several such tool categories but only the low-level ones will be touched here.

A.1 Low-level approaches

debit is the companion software to the paper of Jean-Baptiste Note and Eric Rannaud [50], which aims at netlist reconstruction from bitstreams and bitstream compilation from an XDL description. The authors have worked mostly with Virtex-2 and Spartan platforms, which is reflected in the state of the software. Additional modifications had to me made to debit within the context of this thesis in order to correctly parse Virtex-5 bitstreams as well as partial bitstreams in particular. These changes are work in progress so far.

In early 2011 two very interesting and almost equivalent software projects have been published, BYU RapidSmith and CSU Torc. The goal in both cases is to provide a comprehensive and open source foundation for FPGA design tool research. They enable device exploration and full manipulation of XDL netlist descriptions. Quoting from the RapidSmith website[19]:

> RapidSmith is a research-based, open source FPGA CAD tool written in Java for modern Xilinx FPGAs. Based on XDL, its objective is to serve as a rapid prototyping platform for research ideas and algorithms relating to low level FPGA CAD tools.

Torc is very similar to RapidSmith, only that it is written in C++. This can have some implications for building tools within these frameworks that are targeting an embedded platform, depending on the available resources. While RapidSmith requires a Java environment, Torc requires the boost libraries. Quoting from the Torc project description at[23]:

> Torc is an open-source C++ infrastructure and tool set for reconfigurable computing, released under the GNU GPL 3.0 license. It is suitable for custom research applications, for CAD tool development, and for architecture exploration.

Finally, later in 2011, yet another project has been released called OpenPR[24]. OpenPR is built upon Torc and provides an alternative way of implementing partially

[23] http://torc-isi.sourceforge.net/
[24] http://openpr-vt.sourceforge.net

reconfigurable systems on Xilinx FPGAs. It is based on *fpga_editor* scripts for implementing the higher-level constraint specifications. OpenPR has only been tested preliminarily but clearly provides a promising path for future work in this area.

In this work, debit and RapidSmith have been used extensively, especially for what is termed "architecture exploration" in the Torc description.

A.2 Custom tools

A small set of custom tools, scripts and related documents has been developed during experimentation which have been made available on Github at[25], named *rchw*. This repository includes the following items.

The first two items relate to preparation and testing of VHDL entities for use within a DPR context. *genmake* is a script for generating a makefile within an Integrated Synthesis Environment (ISE) project in order to compile a single component without automatic adding of *iobufs* which is needed for netlists that are later included as implementations of black boxes referenced earlier in the design. An additional option is the inclusion of *ghdl* as a simulation target. ghdl is an open source VHDL simulator which can be used for behavioural simulation of HDL entities. A companion piece of software to ghdl is *gtkwave* which will render the value change dump (.vcd) files produced by a ghdl run. *make* based synthesis of netlists also facilitates template based code generation from outside of VHDL for multiple module implementations. The second item is called *vhdlparse*. This is an ad-hoc attempt at a VHDL parser and was needed for generating wrappers from VHDL entity descriptions as well as generating graphical documentation of entities. The latter function is only preliminarily implemented. vhdlparse can be invoked via the *mode* command-line switch to choose from these two options.

```
user@host ~$ vhdlparse.py -h
Usage: vhdlparse.py [options]

Process HDL code, V1

Options:
  -h, --help            show this help message and exit
  -m MODE, --mode=MODE  set mode: (wrapgen|graphdoc|graphdoc2)
  -f TARGET, --file=TARGET
                        set hdl file)
  -v, --verbose         operate verbously
  -q, --quiet           operate non-verbously
```

The *wrapgen* mode generates a wrapper for a given entity which can then be imported as a pcore into EDK. The *graphdoc** modes generate a graphical representation of the respective entity for documentation purposes. Some example outputs appear throughout

[25] https://github.com/x75/rchw

the text above. Some of these graphics have experienced additional editing. vhdlparse is written using the *pyparsing* Python module[26].

The *rchw* project includes the GA code used for the LUT configuration evolution experiment, described in Genetic Algorithm for intrinsic evolution of LUT configuration above. The GA Python script is built on top of the *pyevolve* Python module[18].

Also included are RapidSmith and Torc code fragments for the specific bitstream manipulations described above. These modules are not generalised in any way and are provided as is, for the purpose of documentation. Fully generic bitstream de- and re-composition would obviously require quite some additional work.

The last sub-collection pertains to application code for the Embedded Linux system, contained in the *elinux* subdirectory. One component is a program for stripping a bitstream of its header and writing the remaining raw data onto the ICAP device. This is derived from Dave Sullins' original *bitinfo* program[27]. Other components are related to audio transmission and the GA evaluation as well as some less specific debugging and probing programs. Also, many hardware modules' HDL is included in the *modules* subdirectory for reference and as a starting point for modifications and extensions. Finally, there is a companion website for this thesis project[28].

In addition, there are several example hardware design projects included. The first one is *audio_reconf_1* and includes the source for an initial version of the reconfigurable audio system. *bitshift* and *clk_reconfig* are two smaller projects focussing on the issues discussed in the section on SDR applications. Finally, *reconf_linux_directbit* is the final version of the DPR testbed inluding audio reconfiguration and several reconfigurable blocks that have been used in a somewhat generic manner to implement and test all of the approaches described above. As an extension necessary to enable direct access to I/O resources, the EDK project from reconf_linux_directbit is included as a component in an ISE project in *xps_import_test*.

[26] http://pyparsing.wikispaces.com/

[27] http://home.earthlink.net/~davesullins/software/index.html

[28] http://www2.informatik.hu-berlin.de/~oberthol/html/Hardware.html

B Dynamic Partial Reconfiguration Microprocessor system

This part is intended as a detailed step-by-step account of setting up a DPR-enable base microprocessor system, such as described in Base system above.

B.1 Ingredients

This section is entirely practical, so in order to be useful the following requirements have to be met:

- A development board with a Virtex-5 featuring a PowerPC processor (The procedure described here is adapted from the Microblaze based Xilinx example of UG744 [81] which is targeting a Virtex-6. That is, anything from Virtex-2 up to Virtex-6 could probably be made to work with modest effort with both hard- and soft-core processors. Conditionally this might also be valid for Spartan-3 and Spartan-6 platforms. The Altera approach has not been considered in any way.
- A working Xilinx toolchain (ISE, EDK and planAhead) with the "PartialReconfiguration" license option.
- Since this is a "moving target", it will be helpful to consider building a static microprocessor system prior or in parallel to a reconfigurable one, as is well described in the Xilinx Open Source Wiki[15], formerly [37]. Two other companion files are the Xilinx Partial Reconfiguration User Guide [79] and the PlanAhead Software Tutorial [81].
- A lot of time.

B.2 Static frame

First the static part of the system is set up, which corresponds to the darker blue area in Figure 9. The microprocessor is intended to handle networking and reconfiguration as well as providing a general execution environment for programs written in C (or another language). The network is the primary interface for any kind of interaction with the device besides a debugging console on the UART port. In terms of periphery this means Memory (DDR2-SDRAM), an Ethernet MAC and a UART port. Additionally, the proxy cores for reconfigurable parts of the system will be added later.

B.2.1 Prelimiaries

As suggested in the relevant Xilinx tutorials [79, 80], a convention about the directory layout is helpful in keeping track of files corresponding to various design stages. Versioning with *git* works well when one goes to the pain of handpicking the necessary files within the hardware build directory or maintains a good .gitignore file. The directory layout suggested and adhered to in this project looks like this:

```
project-toplevel-directory/ # name as you want
 - edk/            # this is where the edk design lives
 - implementation/ # synthesized netlists for Reconfigurable
                   # modules
 - image/          # image data such as:
                   #   bitstreams, sysace files,
                   #   xparameters.h, device-tree,
                   #   linux-kernel.elf, etc)
 - tools/          # tcl-, shell-scripts and other helpers)
 - pa*/            # various stages of planAhead integration
```

This structure can be created readily with the script below but of course any other namings may be chosen.

Listing 7: Shell script for creating directory layout of DPR working directory.

```
#!/bin/sh
proj=reconf_linux
for d in edk implementation image tools ; do
   mkdir -p ${proj}/${d}
done
```

B.2.2 Base system

In the Embedded Development Kit (EDK) use the Base-System Builder (BSB) to set up an initial embedded system design. Place the system.xmp file in the project/edk directory. Set up a basic processor system (PowerPC, Microblaze, both etc depending on the resources) and pick DDR2_SDRAM, Hard_Ethernet_MAC, RS232_UART_1 as well as any other kind of peripherals you would like to use later, on the Peripheral configuration page. Once the wizard is finished, add an ICAP IP core to the project. It is a good idea at this point to verify that the system works by building the bitstream, uploading it to the board and running some test program on the processor in standalone mode.

B.2.3 Setting up Linux

After this verification step, the Linux system can be set up. There are little constraints in the choice of resource but using some Embedded Linux distribution is suggested. There are four basic components which need to be taken care of. This is the *cross-compile environment*, the *file system*, the *Linux kernel* and the system address information contained in a *device-tree* (.dts) file which can be exported from the EDK project.

In this approach, *buildroot* [33] was used. The buildroot environment lets you configure the processor architecture for the cross-compiler as well as the components included in the file system. It will then first build the cross-compiler and finally the target file system including all configuration files as well as system programs and generate an image which can be unpacked onto the target filesystem medium. In the current case,

a Compact Flash card was used as the filesystem medium, but also ramdisk images or other media can be considered. Another attractive option for more intricate setups is the OpenEmbedded [51] meta-distribution which has recently added support for ML507 based microprocessor system templates. buildroot is lightweight and a good option to start with.

Before you can build the kernel a device-tree file has to be created that is used during the kernel build process. A device-tree generator for EDK can be pulled from the Xilinx git repository. The procedure for setting it up is described in the Xilinx Open Source Wiki[29]. The generated file has to be copied to the kernel build directory, for example

```
cp edk/ppc440_0/libsrc/device-tree_v0_00_x/xilinx.dts \
  linux-A.B/arch/powerpc/boot/dts/
```

The Linux kernel of the 2.6 family for the current system was built from the the Xilinx git-tree[30], but a current (3.2) vanilla kernel works as well. The overall process of setting up the necessary structures and more is well described in the Xilinx Open Source Wiki [37] and e.g. in other resources such as [11, 17].

Once you have built the kernel and filesystem, it is time again to verify that everything works on the static hardware built earlier. Configure the hardware with the bitstream from download.bit and load the kernel ELF file with XMD. Boot Linux and verify you have a working base system. The kernel especially needs to be told about which console device to use and the location of the root filesystem.

B.2.4 Enabling reconfigurable cores

After the base system has been verified to boot Linux, the system can be extended with custom IP cores which act as placeholders for RMs which are to be implemented later. The module suitable for testing the dynamic reconfiguration process is a simple blinking LED circuit. The module at this point has no direct connection with the microprocessor and can also be used later to verify that the hardware configuration has been loaded successfully by the initial configuration method. Partial reconfiguration can be tested with the external JTAG based configuration method while the Linux system can be observed remaining in operation across reconfigurations.

While the logic contained in RMs is arbitrary apart from size restrictions, the interface between static and dynamic regions obviously needs to be fixed. In this case, this is done by importing the RM wrapper (blink.vhd) as a custom pcore into EDK and adding that to the main design. The wrapper implements the interface for static the side of the system. It will often be identical to that of the contained RM but that does not need to be so. All signals, except the clock signals, are suggested to be registered in UG702 [79, pp. 37,121]).

> Xilinx strongly recommends that all signals, except global clocks, passing through the Reconfigurable Partition boundary are registered to simplify

[29] http://xilinx.wikidot.com/device-tree-generator
[30] git://git.xilinx.com/linux-2.6-xlnx.git

timing constraints and to increase the likelihood that timing constraints are met. However, if pads are connected (p. 37)

...

It is very important to register the partition boundaries, and to use enables with these registers. During reconfiguration, the activity in these regions is indeterminate and could lead to design corruption if the output of the reconfiguring logic is used. (p. 121)

The following codelisting defines a wrapper for the blink module, whose implementation is given further below.

Listing 8: Blink module wrapper for EDK.

```
-- wrap a very simple blink module for use in an EDK
-- based microprocessor project

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity blink is
  Port(
    clkin : in  STD_LOGIC;                -- clock it
    led : out  STD_LOGIC                  -- single bit LED output
  );
end blink;

architecture Behavioral of blink is
  component blink_act
    port (
      clkin : in  std_logic;                -- clock
      led : out std_logic);                 -- LED driver
  end component;

  signal led_reg : std_logic := '0';    -- LED register

begin
  -- user logic
  blink_instance : blink_act
    port map (
      clkin => clkin,                       -- clock is unregistered
      led => led_reg
    );

  reg_proc : process(clkin)
  begin
    if rising_edge(clkin) then
      led <= led_reg;                       -- transfer to output port
    end if;
```

```
  end process reg_proc;

end Behavioral;
```

The wrapped entities are handled as blackboxes during synthesis and are only need during implementation later on. This entails that any component implementation complying to the same interface can later be plugged into this spot (basically independent of whether this happens within a static or dynamic design target). The wrapper code now needs to be imported as a user pcore into EDK via

```
Hardware -> Create or import peripheral.
```

and then importing an existing peripheral to the XPS project from its respective HDL. When the pcore definition is known to EDK, the block can be placed in the system layout and the ports connected, as usual. When this is completed, synthesis has to be triggered by the usual means by either calling

```
Hardware -> Generate netlists
```

or by issuing

```
$ make netlist
```

from the shell within the edk/ directory. EDK is not able to generate a full implementation of the DPR system. This is delegated to planAhead. The netlist generation command will leave a set of netlists (.ngc files) in the edk/implementation directory which will be used in planAhead later on in order to complete the full build. Before that can be done, the RM implementations need to be defined.

B.2.5 Building reconfigurable modules

This defines the logic for the RMs. A single bit alternating data source is defined with the following VHDL code.

Listing 9: Blink module implementation example.

```
-- Reconfigurable Module implementation for LED blink module
-- high frequency

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.all;

entity blink_fast is
  Port (
    clkin : in  STD_LOGIC;
    led : out  STD_LOGIC
  );
end blink_fast;
```

```
architecture Behavioral of blink_fast is
  signal count : unsigned(31 downto 0) := X"00000000";  -- counter

begin
  process(clkin)
  begin
    if rising_edge(clkin) then
      count <= count + 1;
    end if;
    -- bit 23 at 100MHz ~0.08 seconds
    led <= std_logic(count(22));
  end process;

end Behavioral;
```

Now this block needs to be synthesised as well. One method is to use the entity definition as top-module in a separate ISE project. Select the "Synthesize -XST" process in the design pane, right-click to access the process properties, go to "Xilinx specific options" and unselect the "-iobuf" option. Now the ISE designs can be synthesised and the resulting netlist file can be copied over into the implementation directory, the final .ngc filename corresponding to the component name defined in the wrapper. An example copy command is given below.

```
$ cp modules/blink_fast/blink_fast.ngc \
  implementation/blink_fast/blink_act.ngc
```

This last command is not strictly necessary of course but facilitates keeping track of multiple files later on. At this point, you also might want to create a second module with some different behaviour which can be used later for the reconfiguration test. Create the HDL source within a new ISE project and prepare as described above. Another option is to use existing .xst and .prj files and call xst directly so as not to litter the filesystem with many ISE projects.

B.3 Integration and bitstream generation in planAhead

In the current partially reconfigurable system design flow it is necessary to define physical regions on the chip, so called Reconfigurable Partitions to spatially constrain run-time reconfigurable logic and keep the implementation of the static part from using this area. This can be done with the planAhead tool in the Xilinx PR flow since at least version 12.1. planAhead also lets you add different implementations for a RP and create bitstreams for each configuration.

RMs are fragments of any kind of hardware that will be implemented within the particular partitions defined just above. To make things easier, all modules that are to be potentially loaded into the same partition need to comply to an identical interface, imposed by the surrounding static logic (wrapper). Start planAhead by typing

```
$ planAhead
```

in the shell.

B.3.1 New project

A new project needs to be created in planAhead. In the new project wizard name your project, select the "synthesized netlist" design source option and click the "Set PR Project" checkbox on the second screen to enable Partial Reconfiguration for this project. Then the top-level netlist that has been generated in EDK earlier needs to be selected by navigating to

```
edk/implementation/system.ngc
```

On the next screen, the design's UCF file has to be added. Select the file from the *data* or *implementation* subdirectory in the edk directory

```
edk/data/system.ucf
```

Finally, the target FPGA has to be selected for which the project should be implemented. This of course depends entirely on your situation. Now project creation can be "Finish"-ed by clicking the respective button and you will be dumped in the planAhead (PA) project manager.

If, as is likely, you will be going back to EDK after a first round in PA for further adjustements of the base system, it might make sense to transfer back and forth the .ucf file between PA and EDK, so that changes from EDK are known to PA and partition regions already defined earlier are not lost, as PA dumps this information into AREA_GROUP constraints.

B.3.2 Populating the reconfigurable modules

Next, click the Netlist Design button on the left-hand side of the PA workpane to load the imported design. PA might complain about certain things. It will report "Undefined Modules", which is not surprising. It might also complain about some errors ("Could not create constraint 'IDELAY_VALUE"'), which can be ignored, see this Xilinx forum entry[31]. When the netlist design is opened, click-open the wrapper netlist-components in the netlist tree display, e.g. "blink_0" ($\text{blink}_{0\text{wrapper}}$) from the above example.

Below two directories (Nets, Primitives) right-click on the component instance and choose the "Set partition" command from the menu. Name the module and select the appropriate netlist (.ngc) from the implementation directory. Repeat the addition of RM netlists for as many as there are via the "Add Reconfigurable Module" command.

If right now, or at a later moment, you want to update (load a different netlist) a module, you first have to make it the active RM (right-click on the module itself in the netlist view and pick "Set as active reconfigurable Module ...") and then use the "Update Reconfigurable Module" command from the component instance context-menu.

[31] http://forums.xilinx.com/t5/PlanAhead/PlanAhead-error-on-IDELAY-VALUE-35-constraint/m-p/71750

B.3.3 Defining partitions / physical regions for RMs

When you are done, go to the "Physical constraints" tab in the design pane. For every RM, select the "Draw Pblock Rectangle" command by right-clicking on the instance block and drawing an appropriate rectangle in the device view to the right. It often helps to zoom in on a specific corner of the device prior to drawing the rectangle by hitting Ctrl-r and drawing a zoom-rectangle. The drawn rectangle will remain highlighted in the device view and corresponds to an AREA GROUP entry in the project UCF file. If RMs access I/O logic directly, the physical region has to be chosen so that the respective I/O components are included in the region.

B.3.4 Running DRC

At that point select "Run DRC" from the left-hand side button list. When the DRC dialog opens, hit the "Clear all" button and then select the "Partial Reconfiguration" checkbox. Run the check and watch for any errors.

B.3.5 Setting up the initial configurations

At the bottom part of the PA window, where the console is located, select the "Design runs" tab. Click-activate the config_1 configuration. In the area above various setting tabs for that configuration will appear.

- Name the configuration
- Create the BM strategy as outlined in the UG744 [81] document.
- In the "partitions" tab, select the module instances you want to implement in that configuration for each RP.

When you are done, right click the configuration in the Config runs tab and select Launch runs. You will have to be build the first config that includes the implementation for the static region, so later configuration runs can import that region rather than re-implement it. When the run has finished, select the "Promote partitions" command so that later configurations runs will know about the already implemented regions, most importantly the static region. After promoting partitions, you can start bitstream generation from the configuration item right click menu. The complete and the partial bitstreams will end up in the

```
pa_project_name/pa_project_name.runs/configuration_name/
```

directory.

B.3.6 Building additional configurations

Once, or technically even before, you have an initial implementation, you can add additional configurations with the "Create Multiple Runs" command in the Design runs tab.

Click yourself through the wizard, set up the runs, i.e. import the static partition(s) and implement all the different RMs. On multi-core machines, several configuration runs can easily be run in parallel jobs.

B.4 Recapitulation

To summarise, we now have various different bitstreams as the result of the last command completion in PA. These include, for every configuration run defined in PA a full bitstream that can be used to fully configure the FPGA with the selected module configuration as well as the "partial bitstream" that can be used to partially reconfigure the RPs defined earlier. At this point it is clear that PA can as well be used to create and manage different system configurations in a "static manner", but, this is of course not what we want exactly. The partial bitstreams can be used with any configuration method without modification, such as JTAG.

One practical method of configuring the FPGA is via a SystemACE file. This consists of the configuration bitstream prepended to the code that is to be loaded into the processor for execution. Basically this worked well but led to trouble later on in this particular setup so initial configuration had to be reverted to loading the full bitstream via JTAG and then loading the Linux kernel into memory via XMD. At this point, you could use the conventional configuration method to reconfigure the system while Linux is running in the static region of the FPGA.

The last bit that we want to have working now is the possibility of using the Internal Configuration Access Port (ICAP) from within the on-board Linux to reconfigure the RPs.

B.5 ICAP and Linux

The ICAP is a special hardware resource, see Figure 38, included in the Xilinx Virtex FPGA series since the Virtex 2. Essentially it implements the SelectMAP configuration interface for internal (in-chip) access.

Figure 38: Virtex5 ICAP_X0Y1 resource in PlanAhead device view.

The Xilinx kernel tree comes with the xilinx_hwicap driver but unfortunately this one

is defunct by at least a copy-and-paste error that appeared during addition of the Virtex-5 chip generation. The correct settings can be gleaned from the standalone example code from the Xilinx UG744. Additionally the configuration process as executed by the driver needs to be modified to do single word writes instead of larger bursts. These last two realisations are due to Nicholas Palladino of the RIT cluster lab [53], where a patched version of the driver has been prepared[32].

After recompiling the patched driver and loading it, the system is almost ready. The only thing left wanting are some lines of code that strip the header off the partial bitstream before writing it onto the icap device. This can be done for example with the *bitinfo* program and derivates such as posted in the rchw repository[25]. An example run would be

```
$ ./bitinfo partial_bitfile.bit
```

If the bitstream header is stripped off the bitfile with e.g. stripheader, the resulting raw bitfile can be cp'ed or cat'ed onto the device file.

```
$ ./stripheader < partial_bitfile.bit >partial_bitfile_raw.bit
$ cat partial_bitfile_raw.bit >/dev/icap0
```

[32] This was written in the beginning of 2011 and the situation appears to have improved. Chances are, that the ICAP driver works out fo the box.

C Evolutionary Algorithms

The ideas of artifical evolution were developed independently by several authors during the 1960s with some initial attempts dating back even further. These idea are borrowed from what is known about biological evolutionary processes. The procedure can be regarded as a search or optimisation method, or, alternatively as an open-ended process of continous adaptation. In simple terms, a solution to a problem P is specified through a genotype g, called an individual. In Evolutionary Computation (EC), these genotypes can be strings of bits, fragments of computer programs, strings of real numbers or potentially aggregate objects of any kind. In order to evaluate the quality of one such solution, the genotype has to be mapped onto a phenotype. This mapping can vary widely in complexity. Some examples are program subtrees in GP, bits in FPGA configuration memory, parameters of arbitrary systems in ES, the weights of a neural network. In more biologically plausible setups, the genotype specifies a plan for constructing the phenotype with the possibility of further environmental interaction during *development*. Once the genotype has been transformed into a phenotype, this phenotype is allowed to act in its natural environment. The type of environment depends on the evolutionary system as a whole and can either be a simulated one or a real physical environment with a parameterised artifact behaving dynamically within.

The crucial step in EAs is the design of the fitness function. This function assigns a value (fitness) to the performance of the instantiated individual based on the system's observable variables. So much for a single individual. EAs are generally population based though. In most algorithms, the population's size is fixed. That is, in one cycle of evaluation, a population of individuals called a generation, are all evaluated serially or even in parallel and assigned a fitness number after some finite evaluation time. Once this has been done, evolutionary operators are applied to the genotypes in order to create the next generation. The three major genetic operators are selection, mutation and crossover. The selection operator selects individuals based on a (probabilistic) function of fitness. The mutation operator is a unary operator and randomly changes parts of the genotype according to preset probabilites. The crossover (or recombination) operator re-combines portions of two or more genotypes into a new single genotype. Many variations and elaborations are applicable for almost any stage of this process (genotype to phenotype mapping, operators, dynamically controlled evolutionary parameters, etc) but these basics apply to almost all variants of evolutionary algorithms. There exists a huge body of literature on these topics from both the biological and artificial angle, for example [61, 28, 23, 49, 57].

D Content of the electronic supplement

This document is accompanied by an electronic supplement (Data-DVD) which contains the following items:

- This thesis in PDF format together with LaTeX sources.
- Tar archive of the *rchw* github repository containing tools used in the experiments.
- A tar archive of modifications to the debit project.
- Tar archives of the hardware design projects
  - reconf_linux_directbit.tar.gz: Main experimental project including audio transmission and LUT evolution.
  - Clk_Delay_Test1_PR.tar.gz: IODELAY experiment
  - modules.tar.gz: many modules and small experiments.
  - xps_import_test.tar.gz: ISE project including the reconf_linux_directbit XPS project as a single component in order for enabling RP exclusive access to I/O pins.
- An electronic fulltext bibliography.

List of abbreviations

AD Analog-Digital . . . 18
ASIC Application Specific Integrated Circuit . . . 21
BRAM Block-RAM . . . 16
BSB Base-System Builder . . . 25
BT Bluetooth . . . 23
CF Configuration Frame . . . 16
CLB Configurable Logic Block . . . 16
CPLD Complex Programmable Logic Device . . . 13
DA Digital-Analog . . . 18
DBM Direct Bitstream Manipulation . . . 2
DDR Double Data Rate . . . 38
DLL Dynamically Loadable Librarie . . . 1
DPR Dynamic Partial Reconfiguration . . . 1
DSO Dynamically Shared Object . . . 1
DSP Digital Signal Processing . . . 21
DiffAn Differential Analyzer . . . 14
EA Evolutionary Algorithm . . . 3
EC Evolutionary Computation . . . 85
EDK Embedded Development Kit . . . 25
EHW Evolvable Hardware . . . 71
ER Evolutionary Robotics . . . 8
ES Evolutionary Strategy . . . 8
F+V Fixed Plus Variable Structure Computer . . . 10
FIR Finite Impulse Response . . . 34
FPAA Field-Programmable Analog Array . . . 14
FPGA Field-Programmable Gate Array . . . iii
GALS Globally Asynchronous Locally Synchronous . . . 61
GA Genetic Algorithm . . . 20
GPIO General Purpose Input Output . . . 35
GPS Global Positioning System . . . 23
GP Genetic Programming . . . 8
GSM Global System for Mobile Communications . . . 23

HDL Hardware Description Language 34
HOS Hardware Operating System 18
I/O Input/Output 2
ICAP Internal Configuration Access Port iv
IOB Input/Output Block 28
IP Internet Protocol 2
ISE Integrated Synthesis Environment 73
LAN Local Area Network 23
LFSR Linear Feedback Shift Register 35
LSB Least Significant Bit 41
LUT Look-Up Table 34
MAC Multiply and Accumulate 21
MP Microprocessor 2
NEC Numerical Electromagnetics Code 8
NoC Network-on-Chip 69
OS Operating System 17
PA planAhead 81
PCB Printed Circuit Board 38
PCM Pulse-Code Modulation 31
PE Processing Element 5
PLB Processor Local Bus 26
PLD Programmable Logic Device 2
PPC PowerPC 25
PR Partial Reconfiguration 15
RC Reconfigurable Computing 11
RF Reconfigurable Frame 16
RL Reconfiguration Logic 17
RM Reconfigurable Module 19
RNG Random Number Generator 61
RP Reconfigurable Partition 17
RTOS Real-Time Operating System 69
SDR Software-Defined Radio 2
SEU Single Event Upset 23

List of Figures

List of Tables

Listings

References

[1] W. Ross Ashby. The W. Ross Ashby Digital Archive, Mar 1948. http://www.rossashby.info/copyright.html. Available from: `http://www.rossashby.info/journal/page/2432+01.html`.

[2] W. Ross Ashby. *Design for a Brain*. Chapman and Hall, 1952.

[3] Salih Bayar and Arda Yurdakul. *Dynamic Partial Self-Reconfiguration on Spartan-III FPGAs via a Parallel Configuration Access Port (PCAP)*, volume 8, page 1–10. Citeseer, 2008. Available from: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.133.5908&rep=rep1&type=pdf`.

[4] Joachim Becker. *A Hexagonal Lattice Continuous-Time Field Programmable Analog Array and Filter Synthesis through Genetic Algorithm*. PhD thesis, Department of Microsystems Technology (IMTEK), University of Freiburg, 2007.

[5] Tobias Becker, Wayne Luk, and Peter Y. K. Cheung. Enhancing Relocatability of Partial Bitstreams for Run-Time Reconfiguration. In Kenneth L. Pocek and Duncan A. Buell, editors, *FCCM*, pages 35–44. IEEE Computer Society, 2007.

[6] P. Bellows and B. Hutchings. JHDL - An HDL for Reconfigurable Systems. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, FCCM '98, pages 175–, Washington, DC, USA, 1998. IEEE Computer Society. Available from: http://dl.acm.org/citation.cfm?id=795657.795802.

[7] Oswald Berthold. Hybride Rechenmedien. Website, 2011. Available from: http://www2.informatik.hu-berlin.de/~oberthol/analog/Analog%20Computing.pdf.

[8] J. Bird and P. Layzell. The evolved radio and its implications for modelling the evolution of novel sensors. In *Proceedings of the Evolutionary Computation on 2002. CEC '02. Proceedings of the 2002 Congress - Volume 02*, CEC '02, pages 1836–1841, Washington, DC, USA, 2002. IEEE Computer Society. Available from: http://www.duke.edu/web/isis/gessler/topics/evolved-radio.pdf.

[9] Brandon Blodget, Philip James-Roxby, Eric Keller, Scott Mcmillan, and Prasanna Sundararajan. A self-reconfiguring platform. In *In Proceedings of Field Programmable Logic and Applications (2003*, pages 565–574, 2003.

[10] Christophe Bobda. *Introduction to Reconfigurable Computing*. Springer, 2007.

[11] Martin Brückner. Linux auf FPGA-Hardware zur Bildverarbeitung, 2007. Student research project. Available from: http://www.informatik.hu-berlin.de/~brueckne/studienarbeit.pdf.

[12] Poki Chen, Chun-Yan Chu, Mon-Chau Shie, Zi-Fan Zheng, and Zhi-Yuan Zheng. A Fully Digital Time-Domain Smart Temperature Sensor Realized With 140 FPGA Logic Elements. *IEEE Transactions On Circuits And Systems—I: Regular Papers*, 54(12):2661–2668, Dec. 2007.

[13] Christopher Claus, Florian Helmut Müller, and Walter Stechele. Combitgen: A new approach for creating partial bitstreams in Virtex-II Pro. In *ARCS Workshops*, pages 122–131, 2006. Available from: http://subs.emis.de/LNI/Proceedings/Proceedings81/article4356.html.

[14] Sesh Commuri, V. Tadigotla, and L. Sliger. Task-based Hardware Reconfiguration in Mobile Robots Using FPGAs. *J. Intell. Robotics Syst.*, 49:111–134, June 2007. Available from: http://dl.acm.org/citation.cfm?id=1265201.1265222, doi:10.1007/s10846-007-9131-3.

[15] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Transactions on Computer Aided Design (TCAD)*, 30(4):473–491, April 2011. Invited Keynote, to appear.

[16] S. Corbetta, F. Ferrandi, M. Morandi, M. Novati, M. D. Santambrogio, and D. Sciuto. Two novel approaches to online partial bitstream relocation in a dynamically reconfigurable system. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, pages 457–458, Washington, DC, USA, 2007. IEEE Computer Society. Available from: `http://dl.acm.org/citation.cfm?id=1260993.1262100`, `doi:10.1109/ISVLSI.2007.99`.

[17] Guillaume Dargaud. Embedded Linux on Xilinx ML405 card: A Tutorial, April 2008. Referring to online version of 20101104. Available from: `http://www.gdargaud.net/Hack/Embedded.html`.

[18] Rolf Drechsler and Nicole Drechsler. GAME-HDL: Implementation of Evolutionary Algorithms Using Hardware Description Languages. In Günther R. Raidl, Jean-Arcady Meyer, Martin Middendorf, Stefano Cagnoni, Juan J. Romero Cardalda, David Corne, Jens Gottlieb, Agnès Guillot, Emma Hart, Colin G. Johnson, and Elena Marchiori, editors, *EvoWorkshops*, volume 2611 of *Lecture Notes in Computer Science*, pages 378–387. Springer, 2003.

[19] R. T. Edwards, K. Strohbehn, S. E. Jaskulek, and R. Katz. Analog module architecture for space-qualified field-programmable mixed-signal arrays. In *Proceedings of the 2nd annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD)*, 1999.

[20] Gerald Estrin. Organization of computer systems: the fixed plus variable structure computer. In *Papers presented at the May 3-5, 1960, western joint IRE-AIEE-ACM computer conference*, IRE-AIEE-ACM '60 (Western), pages 33–40, New York, NY, USA, 1960. ACM. Available from: `http://doi.acm.org/10.1145/1460361.1460365`, `doi:http://doi.acm.org/10.1145/1460361.1460365`.

[21] Gerald Estrin. Reconfigurable computer origins: the UCLA fixed-plus-variable (F+V) structure computer. *IEEE Ann. Hist. Comput.*, 24:3–9, October 2002. Available from: `http://dx.doi.org/10.1109/MAHC.2002.1114865`, `doi:http://dx.doi.org/10.1109/MAHC.2002.1114865`.

[22] John R. Koza et al. *Genetic Programming IV - Routine Human-Competitive Machine Intelligence*. Kluwer, 2003.

[23] David B. Fogel. *Evolutionary Computation*. IEEE Press, 2000.

[24] Pauline C. Haddow and Gunnar Tufte. Bridging The Genotype-Phenotype Mapping For Digital FPGAs. In *Evolvable Hardware*, pages 109–115. IEEE Computer Society, 2001.

[25] Pauline C. Haddow and Andy M. Tyrrell. Challenges of evolvable hardware: past, present and the path to a promising future. *Genetic Programming and Evolvable Machines*, 12:183–215, September 2011. Available from: `http://dx.`

doi.org/10.1007/s10710-011-9141-6, doi:http://dx.doi.org/10.1007/s10710-011-9141-6.

[26] Sverre Hamre. Framework for self reconfigurable system on a Xilinx FPGA. Master's thesis, NTNU - Norwegian University of Science and Technology, 2009.

[27] R.W. Hartenstein, A.G. Hirschbiel, and M.Weber. Xputers - An Open Family of Non von Neumann Architectures. In *Proc. of 11th ITG/GI-Conference: Architektur von Rechensystemen*. VDE-Verlag, 1990.

[28] John H. Holland. *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, 1975.

[29] Edson L. Horta and John W. Lockwood. PARBIT: A Tool to Transform Bitfiles to Implement Partial Reconfiguration of Field Programmable Gate Arrays (FPGAs). Available from: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.22.7044.

[30] Sean V. Hum and Robert J. Davies. An evolvable antenna platform based on reconfigurable reflectarrays. In *Proceedings of the 2005 NASA/DoD Conference on Evolvable Hardware*, pages 139–146, Washington, DC, USA, 2005. IEEE Computer Society. Available from: http://dl.acm.org/citation.cfm?id=1078025.1078359, doi:10.1109/EH.2005.9.

[31] C. Richard Johnson Jr. and William A. Sethares. *Telecommunication Breakdown - or How I Learned to Stop Worrying and Love the Digital Radio*. Prentice Hall, 2003. Available from: http://homepages.cae.wisc.edu/%7esethares/telebreak.pdf.

[32] Krzysztof Kępa, Fearghal Morgan, Krzysztof Kościuszkiewicz, Lars Braun, Michael Hübner, and Jürgen Becker. FPGA Analysis Tool: High-Level Flows for Low-Level Design Analysis in Reconfigurable Computing. In *Proceedings of the 5th International Workshop on Reconfigurable Computing: Architectures, Tools and Applications*, ARC '09, pages 62–73, Berlin, Heidelberg, 2009. Springer-Verlag. Available from: http://dx.doi.org/10.1007/978-3-642-00641-8_9, doi:http://dx.doi.org/10.1007/978-3-642-00641-8_9.

[33] Peter Korsgaard. Buildroot: making Embedded Linux easy. http://buildroot.org/, Dec. 2010. Available from: http://buildroot.org/.

[34] Yana Esteves Krasteva, Eduardo de la Torre, Teresa Riesgo, and Didier Joly. Virtex II FPGA Bitstream Manipulation: Application to Reconfiguration Control Systems. In *FPL*, pages 1–4. IEEE, 2006.

[35] David Krutz. *Ein Betriebssystem für konfigurierbare Hardware*. PhD thesis, Humboldt-Universität zu Berlin, 2007. [Online: Stand 2012-01-20T09:44:04Z]. Available from: http://edoc.hu-berlin.de/docviews/abstract.php?id=27784.

[36] Derek S. Linden. Optimizing signal strength in-situ using an evolvable antenna system. In *Evolvable Hardware*, pages 147–151. IEEE Computer Society, 2002.

[37] John Linn, bones, and Julie Zhu. Xilinx Open Source Wiki, Nov. 2010. Available from: `http://xilinx.wikidot.com/`.

[38] Shih-Chii Liu, Jorg Kramer, Giancomo Indiveri, Tobias Delbruck, Rodney Douglas, and Carver A. Mead. *Analog VLSI: Circuits and Principles*. MIT Press, 2002.

[39] Matthias Lorenz. Synthese analoger Filter durch genetische Programmierung - Anwendung auf einer FPAA-basierenden Hardwareplattform. Studienarbeit, 2009.

[40] Philippe Manet, Daniel Maufroid, Leonardo Tosi, Gregory Gailliard, Olivier Mulertt, Marco Di Ciano, Jean-Didier Legat, Denis Aulagnier, Christian Gamrat, Vincenzo La Barba Raffaele Liberati, Pol Cuvelier, Bertrand Rousseau, and Paul Gelineau. An evaluation of dynamic partial reconfiguration for signal and image processing in professional electronics applications. *EURASIP Journal on Embedded Systems*, 2008.

[41] Daniel Mange, André Stauffer, and Gianluca Tempesti. Toward robust integrated circuits: The embryonics approach. In *Proceedings of the IEEE*, pages 516–541, 2000.

[42] Tomás Martínek and Lukás Sekanina. An Evolvable Image Filter: Experimental Evaluation of a Complete Hardware Implementation in FPGA. In *ICES*, pages 76–85, 2005.

[43] Carver Mead. *Analog VLSI and Neural Systems*. Addison-Wesley, 1989.

[44] Michael Menz. Realisierung von global asynchronen und lokal synchronen Strukturen auf programmierbaren Logikschaltungen mit verschiedenen Clock-Domains, 2006.

[45] Melanie Mitchell and Charles E. Taylor. Evolutionary Computation: An Overview. *Annu. Rev. Ecol. Syst.*, 1999.

[46] Federico Nava, Donatella Sciuto, Marco D. Santambrogio, Stefan Herbrechtsmeier, Mario Porrmann, Ulf Witkowski, and Ulrich Rückert. Applying dynamic reconfiguration in the mobile robotics domain: A case study on computer vision algorithms. *TRETS*, 4(3):29, 2011.

[47] Nadia Nedjah and Luiza de Macedo Mourelle. An efficient problem-independent hardware implementation of genetic algorithms. *Neurocomputing*, 71(1–3):88 – 94, 2007. Dedicated Hardware Architectures for Intelligent Systems, Advances on Neural Networks for Speech and Audio Processing. Available from: `http://www.sciencedirect.com/science/article/pii/S0925231207002172`, `doi:10.1016/j.neucom.2006.11.032`.

[48] Stephen Neuendorffer. Generic partially reconfigured processor systems applied to software defined radio. In *Proceeding of the SDR 07 Technical Conference and Product Exposition*, 2007.

[49] Stefano Nolfi and Dario Floreano. *Evolutionary Robotics*. MIT Press, 2000.

[50] Jean-Baptiste Note and Éric Rannaud. From the bitstream to the netlist. In Mike Hutton and Paul Chow, editors, *FPGA*, page 264. ACM, 2008.

[51] OpenEmbedded Developers. Openembedded, the build framework for embedded linux. http://www.openembedded.org/, Jan. 2011. Available from: `http://www.openembedded.org/`.

[52] Björn Osterloh, Harald Michalik, Sandi Alexander Habinc, and Björn Fiethe. Dynamic partial reconfiguration in space applications. In *Proceedings of the 2009 NASA/ESA Conference on Adaptive Hardware and Systems*, AHS '09, pages 336–343, Washington, DC, USA, 2009. IEEE Computer Society. Available from: `http://dx.doi.org/10.1109/AHS.2009.13`, `doi:http://dx.doi.org/10.1109/AHS.2009.13`.

[53] Nicholas Palladino. Wiki for the Computer Engineering department's FPGA cluster at RIT. http://fpgawiki.ce.rit.edu/index.php/Main_Page, Dec. 2010. Available from: `http://fpgawiki.ce.rit.edu/index.php/Main_Page`.

[54] Kyprianos Papadimitriou, Antonis Anyfantis, and Apostolos Dollas. An Effective Framework to Evaluate Dynamic Partial Reconfiguration in FPGA Systems. *IEEE T. Instrumentation and Measurement*, 59(6):1642–1651, 2010.

[55] Gordon Pask. Physical Analogues to the Growth of a Concept. In A. Uttley, editor, *Mechanisation of Thought Processes*, pages 877–922. HMSO London, 1959. Available from: `http://www.pangaro.com/pask/pask%20physical-analogues-to-growth-of-a-concept.pdf`.

[56] Gordon Pask. The Natural History of Networks. In Marshall C. Yovits and Scott Cameron, editors, *Self-organizing systems, Proceedings of an Interdisciplinary Conference*, volume 2 of *International Tracts in Computer Science and Technology and their Application*, pages 232–263. Pergamon Press, 1960. Available from: `http://ia600404.us.archive.org/10/items/SelfOrganizingSystems/SelfOrganizingSystems.djvu`.

[57] Rolf Pfeifer and Josh Bongard. *How the Body Shapes the Way We Think*. MIT Press, 2007.

[58] Jean-Marc Philippe, Benoît Tain, and Christian Gamrat. A self-reconfigurable FPGA-based platform for prototyping future pervasive systems. In *Proceedings of the 9th international conference on Evolvable systems: from biology to hardware*, ICES'10, pages 262–273, Berlin, Heidelberg, 2010. Springer. Available from: `http://dl.acm.org/citation.cfm?id=1885332.1885362`.

[59] Grégory Pierre-Louis and Justin Wells. Temperature Estimation Using Ring Oscillators. Technical report, Worcester Polytechnic Institute, 2009.

[60] Franz J. Rammig. A concept for the editing of hardware resulting in an automatic hardware-editor. In *Proceedings of the 14th Design Automation Conference*, DAC '77, pages 187–193, Piscataway, NJ, USA, 1977. IEEE Press. Available from: `http://dl.acm.org/citation.cfm?id=800262.809125`.

[61] Ingo Rechenberg. *Evolutionsstrategie. Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Frommann Holzboog, 1973.

[62] Thomas W. Rondeau. *Application of Artificial Intelligence to Wireless Communications*. PhD thesis, Virginia Polytechnic Institute and State University, 2007. Available from: `http://scholar.lib.vt.edu/theses/available/etd-10052007-081332/unrestricted/rondeau-dissertation.pdf`.

[63] SDR Forum. SDRF Cognitive Radio Definitions, 2007. Available from: `http://www.wirelessinnovation.org/assets/documents/SDRF-06-R-0011-V1_0_0.pdf`.

[64] D. Sheldon, R. Roosta, M. Sadigursky, and A. Farrokhy. Monitoring Temperature in SRAM-based FPGAs using a Ring-Oscillator Design, 2007. Presentation Slides, JPL and Caltech.

[65] Lee Spector. *Automatic Quantum Computer Programming: A Genetic Programming Approach*. Kluwer, 2004.

[66] Gianluca Tempesti, Daniel Mange, and André Stauffer. A Self-Repairing Multiplexer-Based FPGA Inspired by Biological Processes. Technical report, EPFL, 1998.

[67] Gianluca Tempesti, Andy M. Tyrrell, and Julian F. Miller, editors. *Evolvable Systems: From Biology to Hardware - 9th International Conference, ICES 2010, York, UK, September 6-8, 2010. Proceedings*, volume 6274 of *Lecture Notes in Computer Science*. Springer, 2010.

[68] Adrian Thompson. Evolving electronic robot controllers that exploit hardware resources. In F. Morán, A. Moreno, J. J. Merelo, and P. Chacon, editors, *Advances in Artificial Life: Proc. 3rd Eur. Conf. on Artificial Life (ECAL95)*, volume 929 of *LNAI*, pages 640–656. Springer-Verlag, 1995.

[69] Adrian Thompson. An evolved circuit, intrinsic in silicon, entwined with physics. In Tetsuya Higuchi, Masaya Iwata, and L. Weixin, editors, *Proc. 1st Int. Conf. on Evolvable Systems (ICES'96)*, volume 1259 of *LNCS*, pages 390–405. Springer-Verlag, 1997.

[70] Gunnar Tufte and Pauline C. Haddow. Biologically-Inspired: A Rule-Based Self-Reconfiguration of a Virtex Chip. In Marian Bubak, G. Dick van Albada, Peter M. A. Sloot, and Jack Dongarra, editors, *International Conference on Computational Science*, volume 3038 of *Lecture Notes in Computer Science*, pages 1249–1256. Springer, 2004.

[71] A.M. Tyrrell, G. Hollingworth, and S.L. Smith. Evolutionary Strategies and Intrinsic Fault Tolerance, 2001.

[72] Miguel A. Vega-Rodriguez, Raul Gutierrez-Gil, Jose M. Avila-Roman, Juan M. Sanchez-Perez, and Juan A. Gomez-Pulido. Genetic Algorithms Using Parallelism and FPGAs: The TSP as Case Study. In *Proceedings of the 2005 International Conference on Parallel Processing Workshops*, ICPPW '05, pages 573–579, Washington, DC, USA, 2005. IEEE Computer Society. Available from: `http://dx.doi.org/10.1109/ICPPW.2005.36`, `doi:http://dx.doi.org/10.1109/ICPPW.2005.36`.

[73] John von Neumann. *Theory of Self-reproducing Automata*. University of Illinois Press, 1966. edited and completed by Arthur W. Burks.

[74] John von Neumann. First Draft of a Report on the EDVAC. *IEEE Ann. Hist. Comput.*, 15:27–75, October 1993. Available from: `http://dx.doi.org/10.1109/85.238389`, `doi:http://dx.doi.org/10.1109/85.238389`.

[75] John W. Williams and Neil W. Bergmann. Embedded Linux as a Platform for Dynamically Self-Reconfiguring Systems-on-Chip. In Toomas P. Plaks, editor, *ERSA*, pages 163–169. CSREA Press, 2004.

[76] Xilinx, Inc. *Xilinx UG191: Virtex-5 FPGA Configuration User Guide*, Aug 2009. Available from: `http://www.xilinx.com/support/documentation/user_guides/ug191.pdf`.

[77] Xilinx, Inc. *Xilinx UG347: ML505/ML506/ML507 Evaluation Platform - User Guide*, Oct 2009. Available from: `http://www.xilinx.com/support/documentation/boards_and_kits/ug347.pdf`.

[78] Xilinx, Inc. *Virtex-5 FPGA Data Sheet: DC and Switching Characteristics*, May 2010. Available from: `http://www.xilinx.com/support/documentation/data_sheets/ds202.pdf`.

[79] Xilinx, Inc. *Xilinx UG702: Partial Reconfiguration User Guide*, May 2010. Available from: `http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_1/ug702.pdf`.

[80] Xilinx, Inc. *Xilinx UG743: PlanAhead Software Tutorial - Overview of the Partial Reconfiguration Flow*, 2010. Available from: `http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_4/PlanAhead_Tutorial_Partial_Reconfiguration.pdf`.

[81] Xilinx, Inc. *Xilinx UG744: PlanAhead Software Tutorial - Partial Reconfiguration of a Processor Peripheral*, 2010. Available from: http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_4/PlanAhead_Tutorial_Reconfigurable_Processor.pdf.

[82] Z. Zhu, D.J. Mulvaney, and V.A. Chouliaras. Hardware implementation of a novel genetic algorithm. *Neurocomputing*, 71(1–3):95 – 106, 2007. Dedicated Hardware Architectures for Intelligent Systems, Advances on Neural Networks for Speech and Audio Processing. Available from: http://www.sciencedirect.com/science/article/pii/S0925231207002184, doi:10.1016/j.neucom.2006.11.031.

Erklärung

Ich erkläre, die vorliegende Diplomarbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt zu haben.

Berlin, den 26. April 2012